HUMBOLDT-UNIVERSITÄT ZU BERLIN

Simulationssprachen – Effiziente Entwicklung und Ausführung

Dissertation

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von Dipl.-Inf. Andreas Blunk

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr. Joachim Fischer
2. Prof. Dr. Andreas Prinz
3. Prof. Dr. Lars Grunske

Tag der Verteidigung: 12. Oktober 2018

Zusammenfassung

Simulationssprachen sind in Bezug auf die Unterstützung neuer domänenspezifischer Konzepte mit einer dem Problem entsprechenden prägnanten Darstellung nicht flexibel erweiterbar. Dies betrifft sowohl die Sprache in ihren Konzepten als auch die Unterstützung der Sprache durch Sprachwerkzeuge. In dieser Arbeit entsteht der neue Sprachentwicklungsansatz *Discrete-Event Modelling with Extensibility* (DMX) für die Entwicklung flexibel erweiterbarer Simulationssprachen für domänenspezifische Anwendungsfelder, der eine effiziente Entwicklung der Sprache und eine effiziente Ausführung von Modellen erlaubt. Der Fokus der Arbeit liegt auf der zeitdiskreten ereignisbasierten Simulation und einer prozessorientierten Beschreibung von Simulationsmodellen. Der Ansatz unterscheidet Basiskonzepte, die zur Basissprache gehören, und Erweiterungskonzepte, die Teil von Erweiterungsdefinitionen sind. Es wird untersucht, welche Basiskonzepte eine Simulationssprache bereitstellen muss, so dass eine laufzeiteffiziente Ausführung von prozessorientierten Modellen möglich ist. Die hohe Laufzeiteffizienz der Ausführung wird durch die Konzeption einer neuartigen Methode zur Abbildung von Prozesskontextwechseln auf ein C++-Programm gezeigt. Durch die Spracherweiterung ist die effiziente Ausführung dabei auf Erweiterungskonzepte übertragbar. Der Spracherweiterungsansatz ist nicht auf Simulationssprachen als Basissprachen beschränkt und wird daher allgemein beschrieben. Der Ansatz basiert auf einer Syntaxerweiterung einer Basissprache, die mit einem Metamodell und einer kontextfreien Grammatik definiert ist. Die Ausführung von Erweiterungskonzepten wird durch eine Konzeptreduktion auf Basiskonzepte erreicht. Der Ansatz stellt bestimmte Voraussetzungen an eine Basissprache und erlaubt bestimmte Arten von Erweiterungen, die in der Arbeit untersucht werden. Die Eignung des Anstatzes zur Entwicklung einer komplexen domänenspezifischen Simulationssprache wird an einer Sprache für Zustandsautomaten gezeigt.

Abstract

Simulation languages are not extensible regarding the support of new domain-specific concepts with a concise representation. This includes the concepts of a language as well as the language tools. In this dissertation, the new approach *Discrete-Event Modelling with Extensibility* (DMX) is developed. DMX allows to create flexible domain-specific simulation languages by defining extensions to a base language. The approach allows to develop these languages efficiently and also to execute simulation models in a runtime efficient way. The focus of this dissertation is on process-oriented descriptions of discrete-event simulation models. The approach distinguishes base concepts which are part of the base language and extension concepts which are part of extension definitions. The dissertation investigates the necessary base concepts which should be included in a base simulation language in order to execute process-oriented models efficiently. The high runtime efficiency of executions is achieved by creating a new method for mapping process context switches to a program in C++. The runtime efficiency can be transferred to extension concepts as well. The extension approach is described in a general way because it is not limited to a simulation language as a base language. The approach is based on the syntax extension of a base language, which is defined by a metamodel and a context-free grammar. The execution of extension concepts is achieved by concept reduction to base concepts. The approach has a number of requirements to the base language and allows certain kinds of extensions, which are desribed in the dissertation. The possibility to define a complex domain-specific simulation language is shown by applying the approach to the development of a state machine language.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Problemstellung

Die Simulation[1] ist eine Experimentiermethode mit der die Funktionsweise und die Eigenschaften komplexer Systeme unter Verwendung von ausführbaren Modellen auf Computern analysiert werden können.

Dazu werden Computersprachen verwendet, die sich in ihren Ausdrucksmitteln und in ihren Eigenschaften für die Beschreibung von Simulationsmodellen unterscheiden. Programmiersprachen stellen allgemeine Beschreibungsmittel bereit, die nah an der Funktionsweise des Computers orientiert sind. Dagegen enthalten Simulationssprachen spezifische Ausdrucksmittel für die Modellierung und Ausführung von Simulationen in Raum und Zeit.

Die Anfänge der Wissenschaftsdisziplin zur Modellierung und Simulation reichen weit zurück. Bereits im Jahre 1961 entstand mit GPSS (General Purpose Simulation System) [30] die erste erfolgreiche Computersprache für das Gebiet der zeitdiskreten Simulation. Im Laufe der Zeit gab es viele Entwicklungen auf diesem Gebiet. Dennoch ist dabei eine Grundvoraussetzung immer gleich geblieben. Die Grundlage einer Simulation bildet immer eine Computersprache, in der ein Modell für die Ausführung auf einem Computer formuliert wird. Damit eine Computersprache für die Simulation geeignet ist, muss sie Konzepte für die Nachbildung von Ereignissen und Aktionen in einem System in Raum und Zeit bereitstellen. Modell und Computer bilden zusammen den Simulator. Die Ausführung des Simulators führt dann zu Ergebnissen, die Antworten zur Funktionsweise und zu den Eigenschaften des untersuchten Systems liefern.

Neben GPSS gibt es viele weitere Simulationssprachen. Die Sprachen unterscheiden sich zum einen im Umgang mit der Modellzeit und zum anderen in der Sichtweise aus der ein Modell formuliert wird. Die Modellzeit ist eine globale Zustandsgröße, die in diskreten oder in kontinuierlichen Schritten erhöht wird. Im diskreten Fall kann der Fortschritt durch das Setzen der Modellzeit auf das zeitlich nächste Ereignis entsprechend einer Next-Event-Simulation oder in einer gleichbleibenden festen Schrittweite erfolgen. Im kontinuierlichen Fall folgt die Modellzeit der Berechnung von gewöhnlichen oder partiellen Differentialgleichungen.

Die Modellsichtweise wird heute in die ereignisorientierte und in die prozessorientierte Sicht unterschieden[2]. Prozessorientierte Sprachen besitzen die Besonderheit, dass Ereignisse und Aktionen, die ein bestimmtes Systemelement betreffen, auch zu-

[1] Der Begriff Simulation meint hier die Computersimulation. Computersimulationen werden heute praktisch ausschließlich digital mit Computern durchgeführt. Die in den 1960er Jahren eingesetzte analoge Simulation besitzt heute keine Bedeutung mehr.

[2] Daneben gibt es noch die aktivitätsorientierte Sicht, die ältere nicht-funktionale Simulationssprachen von den prozessorientierten Sprachen abgrenzt.

sammenhängend in Form einer Beschreibung des Lebenslaufs dieses Elementes formuliert werden können. Prozessorientierte Modelle gelten damit allgemein als gut nachvollziehbare Modellierungssicht.

Ein prozessorientiertes Modell lässt sich in verschiedenen Sprachen beschreiben. Es gibt Sprachen von allgemeiner Art, so genannte *Mehrzwecksprachen* (GPLs), die allgemeine Sprachelemente enthalten, und für die Lösung verschiedenster Klassen von Problemstellungen eingesetzt werden können. Zu diesen Sprachen gehören Programmiersprachen. Daneben gibt es Sprachen mit domänenspezifischen Elementen. Diese Sprachen werden heute als *domänenspezifische Sprachen* (DSLs) [25] bezeichnet. In beiden Arten von Sprachen ist es möglich Modelle für eine Simulation zu entwickeln. In GPLs erfolgt die Beschreibung von Simulationskonzepten als Bibliothek in der Sprache. In DSLs sind die Konzepte dagegen in der Sprache integriert.

Die Bezeichnung einer Sprache als GPL oder DSL erfolgt relativ zum Problem. Die Sprache GPSS ist eine DSL für das Gebiet der zeitdiskreten Simulation, wenn als Bezugspunkt eine GPL zur Softwareentwicklung gewählt wird. Sie bietet spezielle Sprachelemente mit denen parallele Prozesse abgebildet werden können und sich z.B. Zufallszahlen verwenden oder Berichte erzeugen lassen. Diese Art von DSL wird auch als *Simulationssprache* bezeichnet. Auf dem Gebiet der zeitdiskreten Simulation stellt GPSS dagegen eine GPL für dieses Gebiet dar, weil mit ihr viele Klassen zeitdiskreter Systeme modelliert werden können.

Im Laufe der Zeit sind viele Sprachen entstanden, mit denen sich Modelle erstellen lassen. Die ersten Sprachen wurden zur Verarbeitung in einem Computer rein textuell notiert[3]. Erst mit der Einführung grafischer Benutzerschnittstellen, entwickelten sich die Beschreibungsmittel weiter und es entstanden auch grafische Sprachen. Heute reicht die Spannweite von Programmiersprachen, die textuell notiert werden und sich flexibel für eine große Bandbreite an Problemen einsetzen lassen, über grafische Modellierungssprachen wie z.B. die Unified Modeling Language (UML) [52], bis hin zu speziell für bestimmte Anwendungsbereiche bzw. Domänen entwickelten Sprachen in Kombination mit Software-Entwicklungsumgebungen, die eine grafische Modellierung und Visualisierung von Simulationen erlauben, ein Beispiel ist SDL-RT [55].

Ein wichtiger Faktor bei der Wahl einer Modellierungssprache ist die prägnante Darstellung von Modellen. In einer Programmiersprache stehen für die Beschreibung eines Modells nur universelle Beschreibungsmittel bereit. Die Modellprägnanz ist deshalb gering. Eine Simulationssprache enthält dagegen Konzepte für eine spezifische Modellklasse, wie z.B. für die Modellklasse der zeitdiskreten ereignisbasierte Systeme. Die Modellprägnanz ist deshalb höher als bei Modellen in einer Programmiersprache. Dennoch sind die Konzepte allgemein und für verschiedene Problemfelder einsetzbar. Eine Simulationssprache für eine spezifische Problemdomäne stellt neben Simulationskonzepten zusätzlich Konzepte aus der Problemdomäne bereit. Ein Beispiel ist die Domäne der reaktiven System, für die UML State Machines eine geeignete DSL darstellen. Die Prägnanz von Modellen in einer Simulations-DSL ist höher als die von Modellen in einer allgemeinen Simulationssprache.

Die bestehenden Modellierungssprachen für die Simulation haben gemeinsam, dass die Sprache aus Sicht des Anwenders unveränderbar durch den Sprachentwickler fest-

[3] Grafische Modelle wurden auf einem Blatt Papier notiert und mussten dann von Hand in eine textuelle Computersprache übersetzt werden. Beispiele hierfür sind die Sprachen CSMP und SDL.

gelegt ist. Diese Festlegung ist ein Hauptproblem aktueller Simulationssprachen. Diese sind zwar hochoptimiert in der Ausführung der Simulatoren, jedoch in Bezug auf die Unterstützung neuer domänenspezifischer Konzepte mit einer dem Problem entsprechenden Darstellung nicht flexibel erweiterbar. Dies betrifft sowohl die Sprache in ihren Konzepten als auch die Unterstützung der Sprache durch Sprachwerkzeuge.

Eine Lösung für dieses Problem stellt eine flexible Simulations-DSL dar, die sowohl allgemeine als auch domänenspezifische Simulationskonzepte enthält und sich außerdem flexibel um neue Domänenkonzepte durch den Modellierer erweitern lässt. Ein Beispiel für eine solche Sprache ist die erweiterbare Simulationssprache *Simulation Language with Extensibility* (SLX). Ein Modell in einer flexiblen Simulations-DSL besitzt eine besonders hohe Prägnanz.

Aus dem Feld der Programmierung entstammt ein ähnlicher Ansatz, der nicht auf eine Anwendung in der Simulation beschränkt ist. Dieser Ansatz wird als *sprachorientierter Programmieransatz* (Language-oriented Programming) [81] [19] bezeichnet. Der Modellierer konzipiert hier vor der Erstellung eines Modells zunächst die DSL mit der Modelle entsprechend der Problemstellung geschrieben werden. Beim sprachorientierten Ansatz wird der Modellierer zunächst zum Sprachentwickler. Er ist dadurch in der Lage, Sprachkonzepte speziell für seine Problemklasse als Teil einer DSL zu definieren, die es erlauben, Lösungen prägnanter und verständlicher zu beschreiben als dies mit einer GPL möglich ist.

Unter den ersten Computersprachen befindet sich mit GPSS bereits eine DSL für die Simulation. Obwohl diese Sprache damals noch nicht mit dem Begriff DSL bezeichnet wurde, gibt es sie bereits so lange wie es allgemeine Programmiersprachen gibt. Die Entwicklung einer Sprache stellt jedoch einen hohen Aufwand dar, da eine Sprache erst durch Sprachwerkzeuge eingesetzt werden kann. Diese Werkzeuge sind Computerprogramme, die Programme in einer Sprache geeignet darstellen oder für die weitere Verarbeitung in andere Repräsentationsformen umwandeln. Die Bereitstellung von Sprachwerkzeugen kann vereinfacht werden, in dem eine Beschreibung ihrer relevanten Aspekte in einer Sprachbeschreibungssprache erfolgt. Die effiziente Entwicklung einer DSL ist daher erst mit weitgehenden Automatisierungen in der Bereitstellung von Sprachwerkzeugen durch Sprachen, die Sprachen beschreiben, möglich.

Damit der sprachorientierte Ansatz praktisch einsetzbar ist, muss die Sprachentwicklung effizient sein. Der Aufwand für die Beschreibung der Sprache einerseits und für die Entwicklung der Sprachwerkzeuge andererseits muss gering sein. Zudem muss die Qualität der Sprachwerkzeuge für eine DSL vergleichbar mit der Qualität von Werkzeugen für Programmiersprachen sein. Dies erreicht der Ansatz durch die automatische Bereitstellung von Sprachwerkzeugen aus Teilaspektbeschreibungen, die in Metasprachen formuliert sind. Dazu gehören Metamodellierungs-Frameworks zur Verarbeitung von Sprachinstanzen in einer Programmiersprache, wie z.B. EMF [24] und AMOF2 [62] sowie Modelleditoren zur Darstellung und Eingabe von Sprachinstanzen in einer gut nachvollziehbaren Notation, wie z.B. TEF [62] und Xtext [36].

Durch die Bereitstellung geeigneter Sprachbeschreibungssprachen und durch die automatisierte Bereitstellung von Sprachwerkzeugen werden die Entwicklungskosten gering gehalten und die Effizienz in der Sprachentwicklung wird erhöht. Eine effiziente Sprachentwicklung ist also charakterisiert durch eine prägnante Sprachbeschreibung

und durch automatisch verfügbare Sprachwerkzeuge.

Eine erste Automatisierung der Sprachwerkzeuge wurde durch die Einführung von Grammatiken erreicht, aus denen Parser für die Verarbeitung von textuellen Sprachen abgeleitet werden können. Ein Teil der Sprache wird nun mit einer Grammatik, die eine speziell geeignete Metasprache für Aspekt der Syntax einer Sprache darstellt, definiert und ein Werkzeug in Form eines Parsers als Programm für Verarbeitung und Übersetzung von Sprachinstanzen kann automatisch erzeugt werden.

Die nächste Stufe der Automatisierung stellen modellbasierte Sprachentwicklungsansätze dar. Jeder Sprachaspekt wird in einer speziellen Metamodellierungssprache beschrieben. Dabei ist der zentrale Sprachaspekt die abstrakte Struktur einer Sprache, die mit einem Metamodell definiert wird. Über dieses Metamodell ist es möglich, Beziehungen zwischen Beschreibungen von Sprachaspekten herzustellen. Das zentrale und strukturgebende Metamodell erlaubt die automatische Bereitstellung weiterer Sprachwerkzeuge, die über das Metamodell miteinander verknüpft sind. Dazu gehören Editoren und Debugger [61] [15].

Der modellbasierte Ansatz erlaubt die Entwicklung beliebiger Sprachen. Es gibt keine bestimmten Voraussetzungen, die eine Sprache erfüllen muss. Diese Freiheit ist jedoch mit dem Aufwand verbunden, jedes erforderliche Grundelement einer Sprache selbst konzipieren zu müssen. Gibt man jedoch die Grundelemente einer Sprache vor, so lässt sich der Aufwand für die Konzeption ähnlicher Sprachen reduzieren und somit die Effizienz in der Sprachentwicklung weiter steigern.

Durch die Vorgabe von Grundelementen, die zwar eine Einschränkung in der Menge konzipierbarer Sprachen bedeuten, erreicht man jedoch auch eine neue Stufe in der Automatisierung von Sprachwerkzeugen. Ansätze, die dieses Prinzip verfolgen, werden als erweiterungsbasierte Ansätze bezeichnet. Dabei wird eine bestehende Ausgangssprache erweitert, um zu einer DSL zu gelangen. Die Möglichkeit ein Metamodell vollkommen frei zu definieren, wird beim erweiterungsbasierten Ansatz bewusst eingeschränkt.

Die ersten theoretischen Untersuchungen zu erweiterbaren Sprachen gab es bereits um das Jahr 1970 [67]. Die damaligen Arbeiten konzentrierten sich auf die Theorie der semantischen und syntaktischen Erweiterbarkeit. Die effiziente Entwicklung stand damals noch nicht im Vordergrund, obwohl sie ebenso wichtig ist. Im Laufe der Jahrzehnte entwickelte sich die Softwaretechnik stetig weiter. Die modellbasierte Softwareentwicklung, die sich auch auf Sprachen anwenden lässt, erlaubte weitreichende Fortschritte in der automatischen Ableitung von Sprachwerkzeugen für domänenspezifische Sprachen. Dazu zählen textuelle Editoren [62], Interpreter auf der Basis einer operationalen Semantik [57] sowie Debugger als besondere Form dieser Interpreter [15]. Die Übertragung des modellbasierten Ansatzes wird aktuell auch für erweiterbare Sprachen untersucht [78] [38]. Es gibt jedoch keine Untersuchungen zur Anwendung auf erweiterbare Simulationssprachen.

Das Besondere erweiterungsbasierter Ansätze besteht darin, dass die allgemeinen oder simulationsspezifischen Elemente der bestehenden Sprache bei einer Verwendung in der DSL nicht neu definiert werden müssen. Die DSL kann diese Elemente wiederverwenden. Dadurch reduziert sich je nach DSL der notwendige Beschreibungsaufwand.

Besonders DSLs, die allgemeinsprachliche Elemente wie z.B. Anweisungen und

Ausdrücke benötigen, profitieren von diesem Ansatz, da deren Entwicklungskosten besonders reduziert werden. Simulationssprachen sind ebenfalls Sprachen mit jeweils ähnlichen Basisskonzepten für die Grundklassen der zeitdiskreten, zeitschrittbehafteten und zeitkontinuierlichen Simulation. Die Effizienz ihrer Entwicklung steigt somit durch den Einsatz von erweiterbaren Sprachen.

Obwohl es auch im Bereich der Simulation eine erweiterbare Simulationssprache gibt [34] [85], sind deren Konzepte nur für die Erweiterung von einfachen Ausdrücken und Anweisungen ausgelegt. Die Syntax von Erweiterungen ist dabei auf reguläre Sprachen beschränkt, die für die Beschreibung von DSLs nicht geeignet sind [10]. Somit gibt es bislang keinen erweiterungsbasierten Ansatz für die Entwicklung einer DSL für die Simulation.

Der Einsatz einer erweiterbaren Sprache hat einen interessanten Nebeneffekt. Durch die Abbildung von Erweiterungen auf die Basissprache, profitieren Erweiterungen automatisch von einer laufzeiteffizienten Implementierung der Basissprachkonzepte. Das ist besonders für Simulationssprachen wichtig, da mit laufzeiteffizient ausführbaren Modellen besonders viele Modellvarianten in kurzer Zeit analysiert werden können.

Eine prozessorientierte erweiterbare Simulationssprache bietet Vorteile im Hinblick auf eine effiziente Sprachentwicklung und gut nachvollziehbare Modelle. Für die laufzeiteffiziente Ausführung sind besonders Kontextwechsel zwischen Prozessen von Bedeutung. Die Arbeit untersucht daher Möglichkeiten zur Implementierung eines hochlaufzeiteffizienten Simulationskerns, der das Laufzeitsystem für die Basissprache bildet.

Zusammenfassend betrachtet erlauben erweiterbare Sprachen eine besonders effiziente Entwicklung von Sprachen durch die Möglichkeit der Wiederverwendung von Basiskonzepten in einer DSL als Teil einer Spracherweiterung. Metamodellbasiert definierte Sprachen erlauben eine effizientere Entwicklung durch die automatische Ableitung von Sprachwerkzeugen und durch eine effiziente Beschreibung von Sprachteilaspekten in DSLs für die Beschreibung von Sprachen.

1.2 Ziel der Arbeit

Das Ziel der Arbeit ist die Konzeption des neuen Sprachentwicklungsansatzes *Discrete-Event Modelling with Extensibility* (DMX), der es erlaubt domänenspezifische Simulationssprachen (Simulations-DSLs) flexibel durch die Erweiterung einer Simulationsbasissprache zu entwickeln. Die Konzeption des Ansatzes erfolgt unter den Nebenbedingungen einer effizienten Entwicklung der Sprache und einer effizienten Ausführung von Modellen in der Sprache. Dazu werden bestehende metamodellbasierte und spracherweiterungsbasierte Entwicklungsansätze kombiniert.

DMX setzt eine Simulationssprache als Basissprache voraus. Die Konzepte des Ansatzes sind jedoch allgemeiner und nicht auf die Domäne Simulation beschränkt. Je nach Domäne kann eine andere passende Basissprache eingesetzt werden, sofern diese bestimmte Voraussetzungen erfüllt. Die Arbeit untersucht die Übertragbarkeit des Ansatzes auf andere Basissprachen und stellt den Ansatz unabhängig von der Domäne Simulation vor. Die Arbeit besitzt dennoch einen starken Bezug zur Simulation und wendet den Ansatz für eine Simulationsbasissprache an, die als *Discrete-Event Ba-*

se Language (DBL) bezeichnet wird. Die Sprache DBL ist flexibel erweiterbar und enthält Konzepte für die Modellierung und Simulation.

Die Effizienzsteigerung wird durch die Entwicklung einer Simulations-DSL als Erweiterung einer Simulationsbasissprache erreicht. Die Arbeit beschränkt sich auf die Sprachaspekte textuelle Syntax und Ausführungssemantik und setzt eine metamodellbasierte Definition der Basissprache voraus. Dabei ist die abstrakte Syntax mit einem objektorientierten Metamodell und die konkrete Syntax mit einer kontextfreien Grammatik definiert. Durch den Spracherweiterungsansatz wird die Basissprachdefinition in den Sprachaspekten abstrakte und konkrete Syntax erweitert.

Das Besondere am Ansatz DMX ist eine Zurückführung von Erweiterungen auf Basiskonzepte mit einem Erweiterungskonzept, das aus vier Komponenten besteht:

1. einer objektorientierten Basissprache,
2. einer Sprache für die Erweiterungsdefinition in der Basissprache,
3. einer Vorschrift für die Abbildung der Syntaxdefinition einer Erweiterung auf die Syntaxdefinition der Basissprache und
4. einem Semantikbeschreibungskonzept, das die Semantik als eine Ersetzung einer Erweiterung durch Basiskonzepte definiert.

Der Ansatz lässt sich auf andere Basissprachen übertragen, die bestimmte Voraussetzungen in Bezug auf ihre Definition (Metamodell und Grammatik) und ihre Konzepte erfüllen.

Die effiziente Entwicklung wird im Hinblick auf eine Aufwandsreduzierung in der Definition einer Simulations-DSL und für den Erhalt von Sprachwerkzeugen betrachtet. Durch den gewählten Ansatz werden Erweiterungen immer auf Basiskonzepte zurückgeführt. Damit können Sprachwerkzeuge, die für die Basissprache bereits implementiert sind, für Erweiterungen wiederverwendet werden. Die Arbeit implementiert einen adaptiven Basismodelleditor, der sich an Erweiterungen anpasst und diese automatisch unterstützt. Des Weiteren wird ein Compiler für Erweiterungen implementiert, der ein Modell mit Erweiterungen auf ein Modell mit Basiskonzepten zurückführt. Das resultierende Basismodell kann dann mit einem Compiler für eine beliebige Zielsprache übersetzt und ausgeführt werden.

Als Basissprache wird eine prozessorientierte Simulationssprache gewählt, da sich Ereignisse, die dem Lebenslauf eines bestimmten Systemelementes zugeordnet werden können, in einem prozessorientierten Modell besonders nachvollziehbar beschreiben lassen. Werden die Elemente einer Domäne als Sprachkonzepte definiert, so lässt sich deren Semantik, als eine Abbildung auf prozessorientierte Konzepte, auf die gleiche Art und Weise nachvollziehbar beschreiben. Damit reduziert sich der Aufwand für die Definition einer Simulations-DSL durch die Verwendung einer prozessorientierten Simulationsbasissprache für die Beschreibung der Semantik von Domänenkonzepten.

Die effiziente Ausführung beschränkt sich auf eine Untersuchung der Laufzeit von Prozesskontextwechseln, da diese einen besonders hohen Einfluss auf die Laufzeit einer prozessorientiert beschriebenen Simulation besitzen und ihre laufzeiteffiziente Implementierung ein offenes Problem darstellt. Dabei wird C++ als Implementierungssprache betrachtet, da die Sprache in Bezug auf laufzeiteffizient ausführbare Programme entworfen ist und einen Compiler mit automatischer Code-Optimierung besitzt.

Beide Aspekte sind für die Laufzeit eines Simulationsmodells relevant. Darüber hinaus besitzt C++ keine Koroutinen, die für die Implementierung eines prozessorientierten Modells eingesetzt werden könnten. Hier besteht die Herausforderung in der Beschreibung einer Implementierung für die laufzeiteffiziente Umsetzung von Koroutinen in einer Sprache, die kein Koroutinenkonzept enthält.

Die Arbeit betrachtet als Zielsprachen sowohl Java als auch C++. Für Java sind metamodellbasierte Sprachwerkzeuge vorhanden, die als Grundlage für die erweiterungsbasierte Entwicklung einer DSL eingesetzt werden können. Für C++ sind dagegen besonders laufzeiteffizient ausführbare Simulationsmodelle möglich. Für die Unterstützung beider Sprachen wird deshalb eine Basissprache entworfen, die sich auf Java und auf C++ abbilden lässt.

Als Teil der Konzeption des Ansatzes bestehen eine Reihe von besonderen Herausforderungen:

1. Der Umfang der Simulationsbasissprache ist so zu wählen, dass der Sprachkern möglichst klein ist und dennoch die Realisierung möglichst laufzeiteffizienter Simulationen erlaubt. Ein kleiner Sprachkern wird angestrebt, da dieser zu weniger Konflikten bei der Definition von Erweiterungen führt.

2. Die Implementierung eines adaptiven Modelleditors für die Basissprache, der Erweiterungen erkennt und diese auf Änderungen der Basissprachdefinition zurückführt.

3. Die Bestimmung der notwendigen Voraussetzungen für eine Übertragung des Ansatzes auf eine beliebige Basissprache.

4. Die Untersuchung der Erweiterungsmöglichkeiten, die der Ansatz durch seine metamodellbasierte Sprachdefinition bietet.

5. Die Beschreibung eines Vorgehens für die Implementierung von laufzeiteffizienten Prozesskontextwechseln in einer prozessorientierten Simulationsbasissprache.

Die Konzeption eines derartigen Ansatzes wurde bislang nicht untersucht. Nach einem Studium der Weltkonferenz[4] zur Simulation ist bislang nicht bekannt, inwiefern sich existierende erweiterungsbasierte und metamodellbasierte Sprachentwicklungsansätze kombinieren lassen. Die Arbeit untersucht daher bestehende Ansätze und nimmt bei Bedarf Anpassungen und Erweiterungen vor. Dazu werden entsprechende Prototypen entwickelt.

1.3 Annahmen und Einschränkungen

Ich gehe davon aus, dass die Betrachtung rein textueller Sprachen relevant ist. Sieht man auf die aktuelle Entwicklung in der Definition von DSLs, so stellt man fest, dass

[4] Die *Winter Simulation Conference* ist die Weltkonferenz zur Simulation. Sie ist nach Teilnehmerzahl die größte Konferenz und findet bereits seit den 1960er Jahren statt. Auf der Webseite http://informs-sim.org befindet sich ein Archiv aller Veröffentlichungen seit 1968. Dabei findet sich als einziger erweiterungsbasierter Ansatz die Sprache SLX. Das Stichwort DSL wird nur in 0,5% aller Einträge erwähnt.

sich viele Entwicklungsumgebungen auf textuelle Sprachen konzentrieren. Dazu gehören MPS [38] [77], Xtext [36] und Spoofax [39]. Dies liegt zum Teil daran, dass die Beschreibung von textuellen Sprachen kostengünstiger ist und diese außerdem bei bestimmten Konzepten wie z.B. Ausdrücken die bessere Repräsentationsform darstellen.

Betrachtet man das Umfeld von UML, eine der am weitesten verbreiteten grafischen Modellierungssprachen, so gibt es auch hier Bestrebungen für bestimmte Teile der Sprache eine textuelle Repräsentation einzuführen. So beschreibt die OMG-Spezifikation *Action Language for Foundational UML* (ALF) [49] eine textuelle Repräsentation für Aktivitätsdiagramme, die im normalen UML nur eine grafische Repräsentation besitzen.

Ich beschränke mich in meiner Arbeit auf die Syntax und die dynamische Semantik von Simulationssprachen. Die statische Semantik, die die Menge von gültigen Programmen in einer Sprache einschränkt, ist nicht Teil meiner Arbeit. Des Weiteren ist die Konzeption auf zeitdiskrete prozessorientierte Simulationssprachen beschränkt.

1.4 Aufbau

Abbildung 1.1 zeigt den Aufbau der Arbeit und die Abhängigkeiten zwischen einzelnen Teilen der Arbeit. Ich erkläre nun mein weiteres Vorgehen aus dem sich der Aufbau der Arbeit herleitet.

Kapitel 2 beschäftigt sich grundlegend mit der Entwicklung und Ausführung von domänenspezifischen Simulationssprachen. Zunächst werden die verschiedenen Arten von Simulationssprachen und ihre speziellen Anforderungen in Bezug auf Konzepte und Laufzeit zusammengefasst. Danach betrachte ich DSLs und beschreibe die Besonderheiten einer Simulations-DSL. Ich erläutere die bestehenden Klassifikationen für Simulationssprachen und DSLs und ordne den Ansatz entsprechend ein. Schließlich beschreibe ich die Konzepte einer metamodellbasierten Sprachentwicklung, die grundlegend für den Aspekt der effizienten Entwicklung von Simulationssprachen sind. Kapitel 2 erläutert auch die Entscheidung für eine prozessorientierte Simulationsbasissprache und für die metamodellbasierte Entwicklung von DSLs als Grundlage des Ansatzes DMX.

Der Ansatz erlaubt die Definition der Konzepte der Simulationssprache auf einer von mehreren Ebenen, die Einfluss auf die laufzeiteffiziente Ausführung haben. Ein Konzept kann in der Basissprache, als Erweiterung oder durch die Anbindung einer bestehenden Bibliothek definiert werden. Daraus ergibt sich die Frage, welchen Umfang die Simulationsbasissprache besitzen sollte, so dass der Basissprachkern möglichst klein ist und Simulationen möglichst effizient sind. Kapitel 3 beschäftigt sich mit dieser grundlegenden Frage und diskutiert den Umfang der Basissprache. Es wird erläutert, welche Konzepte die Basissprache enthält und an welchen Stellen Entscheidungsalternativen existieren.

Kapitel 3 konzipiert die prozessorientierte Simulationssprache DBL, die einen möglichst kleinen Sprachkern besitzen soll, der gleichzeitig möglichst laufzeiteffiziente Simulationen erlaubt. Dazu werden die Konzepte einer prozessorientierten Simulationssprache im Hinblick auf die Definition als Basiskonzept und als Erweiterung untersucht. Als Diskussionsgrundlage dienen die Programmiersprachen Java und C++

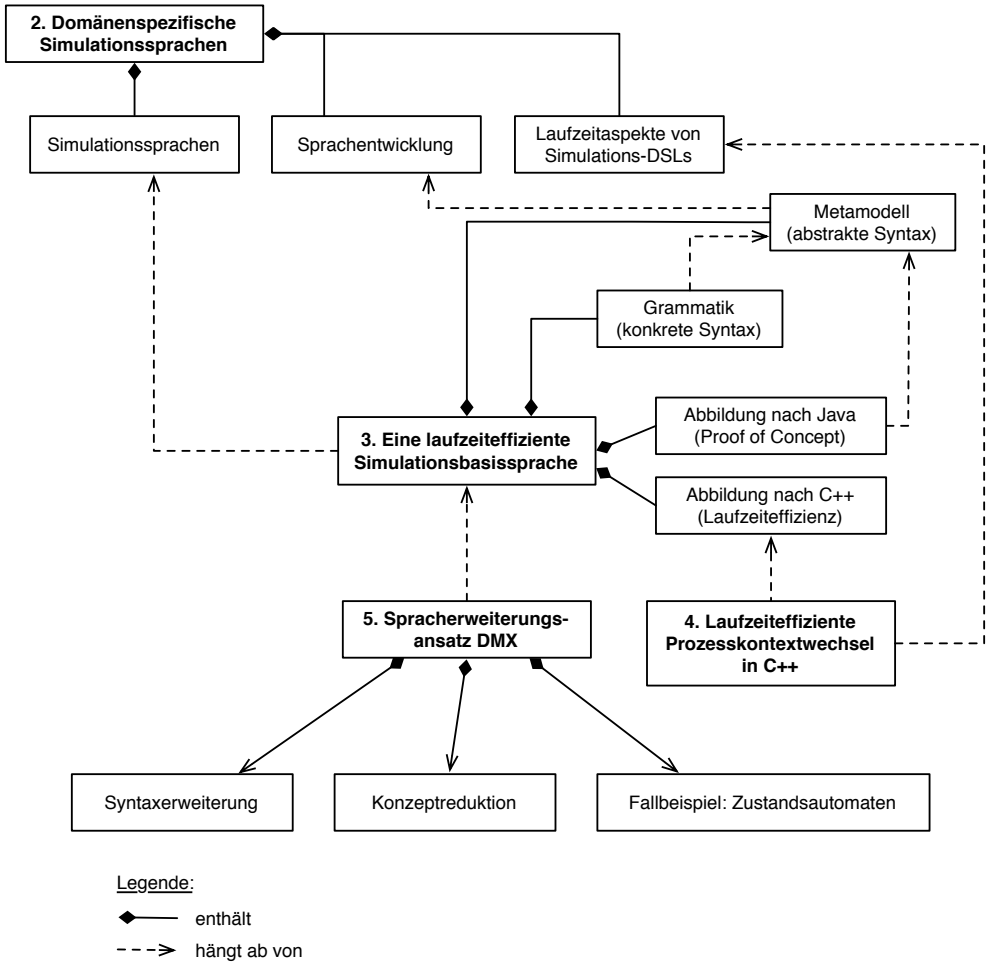

Abbildung 1.1: Aufbau der Arbeit und Abhängigkeiten zwischen einzelnen Teilen der Arbeit.

sowie die einzig bekannte erweiterbare Simulationssprache SLX. Danach erfolgt eine Zusammenfassung der gewählten Basiskonzepte für DBL sowie eine Beschreibung der abstrakten Laufzeitkonzepte und ihrer Semantik. Im Anschluss wird ein Schnittstellenkonzept vorgestellt, mit dem sich bestehende Java- und C++-Bibliotheken in einem DBL-Programm wiederverwenden lassen. Den Abschluss bildet eine Zusammenfassung von Basissprachen in vergleichbaren Sprachentwicklungsansätzen.

Kapitel 4 beschäftigt sich mit der laufzeiteffizienten Realisierung von Prozesskontextwechseln in einer Sprache ohne eine Koroutinenkonzept am Beispiel von C++. Es wird eine Methode beschrieben, mit der ein prozessorientiertes Simulationsmodell auf ein C++-Programm abgebildet werden kann, das eine sehr hohe Laufzeiteffizienz bei der Durchführung einer Simulation besitzt. Dazu wird die Emulation von Koroutinen und eines Funktionsaufruf-Stacks in C++ beschrieben. Der Nachweis einer besonders hohen Laufzeiteffizienz erfolgt durch Laufzeitvergleiche mit bestehenden Simulationsbibliotheken.

Im Kapitel 5 erfolgt die Beschreibung des Erweiterungsansatzes DMX. Zunächst wird ein Überblick zu den vier Komponenten des Ansatzes und ihrer Implementierung gegeben. Danach werden die Komponenten einzeln betrachtet. Zuerst wird die Definition der Basissprache erläutert und das Sprachkonzept für die Erweiterungsdefinition vorgestellt.

Danach wird die Abbildung der Syntaxdefinition von Erweiterungen auf die Syntaxdefinition der Basissprache beschrieben. Die Abbildungsvorschrift dient als Grundlage für einen adaptiven Modelleditor, dessen prototypische Implementierung vorgestellt wird. Es werden Voraussetzungen an die Definition der Basissprache identifiziert, so dass diese ausgetauscht werden kann. Im Anschluss wird die Konzeptreduktion erklärt, die Erweiterungen auf Basiskonzepte zurückführt. Der Erweiterungsansatz bietet bestimmte Erweiterungsmöglichkeiten, die im nachfolgenden Abschnitt betrachtet werden. Als Anwendungsbeispiel folgt die Einführung einer Simulations-DSL für die Modellierung mit Zustandsmaschinen, das die grundlegende Realisierbarkeit einer komplexen Simulations-DSLs mit dem Ansatz DMX zeigt. Abschließend erfolgt ein Vergleich von DMX mit ähnlichen Ansatz in Bezug auf eine effiziente Entwicklung und Ausführung von Simulations-DSLs.

Die Arbeit endet in Kapitel 6 mit einer Zusammenfassung der Beiträge und einem Ausblick für weitergehende Forschungen.

2 Domänenspezifische Simulationssprachen

In diesem Kapitel erfolgt zunächst eine grundlegende Auseinandersetzung mit den Besonderheiten von Simulationssprachen und den Möglichkeiten ihrer effizienten Entwicklung in Form von domänenspezifischen Sprachen für die Simulation. Dazu werden grundlegende Begriffe aus den Bereichen Simulation und Sprachentwicklung eingeführt. Insbesondere wird geklärt, was die Merkmale von domänenspezifischen Simulationssprachen und deren spezielle Anforderungen an eine effiziente Modellentwicklung und Modellausführung sind. Es werden vorhandene Möglichkeiten für eine effiziente Sprachentwicklung vorgestellt. Dabei werden die speziellen Laufzeitaspekte von Simulationssprachen erörtert, die bei der Konzeption des Ansatzes zu berücksichtigen sind.

2.1 Simulation

Mit Simulation bezeichnet man das zielgerichtete Experimentieren durch einen Beobachter, der in Bezug auf ein zu untersuchendes Phänomen, eine Erkenntnis über eine bestimmte Fragestellung mit Hilfe eines Modells zu erlangen versucht. Ein Modell ist ein eingeschränktes Abbild eines relevanten Ausschnitts eines betrachteten Systems. Die relevanten Elemente eines Modells ergeben sich stets aus der Fragestellung, die mit dem konkreten Modell beantwortet werden soll. Es gibt also häufig nicht das eine Modell eines Systems, das für alle möglichen Fragen einsetzbar ist. Im Allgemeinen kann es daher viele Modelle eines Systems geben, die unterschiedliche oder gleiche Fragestellungen betrachten.

Die in dieser Arbeit betrachteten Modelle sind Computermodelle. Der Modellierer entwickelt ein Computerprogramm, das Beschreibungen von Zustandsgrößen und Aktionssequenzen in Abhängigkeit einer Modellzeit enthält. Die Modellzeit ist eine globale Zustandsgröße, deren Wertebereich diskret oder kontinuierlich sein kann und die vom Simulationssystem verwaltet wird. Durch die Abarbeitung der Aktionen in der Modellzeit werden die Werte der Zustandsgrößen verändert. Diesen Vorgang bezeichnet man als Simulation. Das betrachtete System wird durch die Ausführung eines Computerprogramms simuliert. Die sich verändernden Zustandsgrößen werden während der Simulation oder an deren Ende ausgeben, um eine Antwort auf die formulierte Fragestellung zu finden.

Auf einem Computer besteht ein Modell aus einem Programm, das alle notwendigen Bestandteile für die Erzeugung des Modells im Speicher des Computers enthält. Obwohl das eigentliche Modell aus dem gesamten Computersystem inklusive seines Speichers besteht, bezeichnet man als Modell häufig nur das Programm selbst, das eine abstrakte Beschreibung des Modells darstellt.

Im Allgemeinen definiert ein Programm dabei eine Menge möglicher Modelle und muss als Modellbeschreibung bezeichnet werden. Da aus einer Modellbeschreibung das Modell jedoch immer automatisch abgeleitet werden kann, wird der Modellbegriff in der Literatur häufig mit der Modellbeschreibung gleichgesetzt. Sofern eine eindeutige Unterscheidung nicht notwendig ist, verwende ich die beiden Begriffe ebenfalls synonym.

Das System als Vorlage für ein Modell bezeichnet in diesem Zusammenhang ein Phänomen, das bestimmte Eigenschaften besitzt. Das Phänomen muss eine bestimmte Funktion erfüllen, den Systemzweck. Dieser muss von einem Beobachter erkennbar sein. Das Phänomen muss außerdem aus anderen Phänomenen zusammengesetzt sein, die durch ihre Wechselwirkungen die Funktion des Systems erbringen. Die Phänomene, aus denen ein System zusammengesetzt ist, werden als Systemelemente bezeichnet. Als dritte Eigenschaft enthält ein System nur die relevanten Systemelemente. Wird also eines der Systemelemente aus dem System entfernt, so kann der Systemzweck nicht mehr erbracht werden.

2.2 Simulationssprache

Die computergestützte Simulation setzt immer eine Modellbeschreibung voraus. Eine Modellbeschreibung wird vom Beobachter des Systems erstellt und muss von ihm selbst oder anderen Personen verstanden werden. Da eine Modellbeschreibung maschinenverarbeitbar sein muss, folgt sie einem Regelwerk, das durch eine Sprache festgelegt ist, in diesem Fall eine Simulationssprache. Die Verständlichkeit einer Modellbeschreibung steht also in direktem Zusammenhang zur verwendeten Sprache.

Wenn die Sprache Konzepte definiert, die sich mit wenig Aufwand durch Beobachtung des Systems in das Modell und umgekehrt auch durch Betrachtung des Modells in das System übertragen lassen, so spricht man von einer prägnanten Modellbeschreibung. Eine prägnante Modellbeschreibung ist im Allgemeinen erstrebenswert, da sie den Aufwand für die Modellierung reduziert und die erstellten Modelle besser verstanden werden.

Damit eine Modellbeschreibung prägnant ist, muss die passende Sprache bereitstehen. Das ist im Allgemeinen aber nicht der Fall. Programmiersprachen, die sehr häufig verwendet werden, sind allgemeine Sprachen, die sich für jedes beliebige Problem einsetzen lassen, und deren Programme als Folge der allgemeinen Beschreibungskonzepte eher wenig prägnant sind. Es gibt jedoch verschiedene Abstufungen anderer spezieller Sprachen, deren Einsatz zu einer bestimmten Ausprägung der Prägnanz führt:

- Auf der ersten Stufe befinden sich die Programmiersprachen, für eine Simulationsbibliothek vorhanden ist,
- die zweite Stufe bilden Simulationssprachen, bei denen Simulationskonzepte als Elemente in die Sprache integriert sind,
- und auf der dritten Stufe befinden sich domänenspezifische Sprachen für die Modellierung und Simulation in einem bestimmten Anwendungsgebiet.

2.2.1 Einteilung nach der Art des untersuchten Systems

Auf allen drei Stufen ist eine weitere Einteilung nach der Art des untersuchten Systems möglich. Richard Nance hat die historische Entwicklung und die spezifischen Merkmale von Simulationssprachen untersucht [46]. Er unterteilt die Simulation[1] in drei Bereiche:

1. diskrete ereignisbasierte Simulation,
2. kontinuierliche Simulation und
3. Monte-Carlo-Simulation.

Die diskrete ereignisbasierte Simulation beruht auf einem mathematischen Modell, das es erlaubt Zustandsänderungen eines Systems zu festgelegten Zeitpunkten auszudrücken und nachzubilden. Typische Anwendungen sind z.B. Modelle zur Nachbildung von Warteprozessen in einer Bank.

Die kontinuierliche Simulation verwendet Modelle aus mathematischen Gleichungen, die Zustandsübergänge in Abhängigkeit der Zeit ausdrücken. Damit lässt sich die kontinuierliche Veränderung von Zustandsgrößen, z.B. der Zulauf von Wasser in einer Badewanne, gut beschreiben.

Analoge Computer, die kontinuierliche Zustandsänderungen durch Kombination bestimmter elektronischer Bauteile exakt nachbilden können, haben heute keine praktische Bedeutung mehr. Der digitale Computer arbeitet dagegen zeitdiskret. Deshalb werden kontinuierliche Simulationen immer durch eine Vielzahl möglichst kleiner zeitdiskreter Zustandsänderungen abgebildet. Letztendlich ist jede kontinuierliche Simulation auf einem digitalen Computer eine zeitdiskrete Simulation.

Sowohl in der diskreten als auch in der kontinuierlichen Simulation wird die Zeit im betrachteten System durch eine Modellzeit nachgebildet. Abläufe deren Zustandsänderungen voneinander abhängen müssen in Bezug auf die Modellzeit synchronisiert werden. Um den Synchronisierungsaufwand gering zu halten, erfolgt die Nachbildung häufig durch eine sequentielle Abarbeitung, der im realen System parallel auftretenden Prozesse. Die Modellzeit wird erhöht, sobald die sequentielle Abarbeitung aller parallelen Prozesse mit Zustandsänderungen zum jeweils aktuellen Zeitpunkt abgeschlossen ist. Die Art der Abarbeitung wird daher auch als quasi-parallele Ausführung bezeichnet. Die Zeit, die während der Abarbeitung vergeht ist die Ausführungszeit auf dem Computer. Eine Simulation ist umso effizienter je geringer ihre Ausführungszeit ist.

Die Monte-Carlo-Simulation ist anders, soll aber zur Vollständigkeit noch erwähnt werden. Sie verwendet rein stochastische Prozesse bei denen die Zeit nicht betrachtet wird. Die Analyse erfolgt rein numerisch auf Basis der Wahrscheinlichkeitstheorie. Ein Beispiel ist der Poisson-Prozess, dessen Zuwächse poisson-verteilt sind und dabei keinen Bezug zur Zeit besitzen.

[1] Simulation wird häufig nicht nur als den Vorgang der Experimentdurchführung bezeichnet, sondern auch als das Wissenschaftliche Forschungsgebiet, das sich mit der Simulation beschäftigt.

2.2.2 Einteilung nach der Art der Implementierung

Nance unterteilt Simulationssprachen weiter nach der Art der Implementierung von Sprachkonzepten:

1. Präprozessor-basierte Erweiterung einer anderen Sprache,
2. Programmbibliothek in einer Programmiersprache und
3. konventionelle Sprache.

Bei einer präprozessorbasierten Erweiterung werden Teile des Modells in einer Präprozessor-Sprache ausgedrückt, die mit Hilfe eines Präprozessors in ein Programm der eigentlichen Sprache übersetzt werden. Eine Präprozessor-Sprache erlaubt nur einfache Wertersetzungen und bietet daher kaum Möglichkeiten für die Formulierung von Konzepten, die zu prägnanten Modellbeschreibungen führen.

Eine Programmbibliothek stellt Teile eines Modells oder einer Simulation wiederverwendbar bereit und reduziert dadurch den Aufwand für die Simulation. Die Prägnanz eines Modells hängt von den Abstraktionsmitteln für die Beschreibung wiederverwendbarer Modellierungskonzepte der Programmiersprache ab.

Eine konventionelle Sprache ist eine Sprache, die Modellierungskonzepte, die zu prägnanten Modellbeschreibungen führen, enthält. Die Sprache ist so gebaut, dass sie auf eine bestimmte Menge von Problemen passt. Der Einsatz einer konventionellen Sprache erlaubt also sehr prägnante Modellbeschreibungen.

2.2.3 Einteilung nach der Modellperspektive

Neben der Implementierung gibt es drei Arten von Perspektiven aus denen ein Modell beschrieben werden kann. Diese werden als Weltansichten bezeichnet und wurden 1962 von Lackner eingeführt [42]. Es werden drei Weltansichten unterschieden:

1. ereignisorientiert,
2. aktivitätsorientiert und
3. prozessorientiert.

Wird jedes Ereignis einzeln und isoliert von anderen Ereignissen beschrieben, so bezeichnet man dies als ereignisorientierte Modellierung. Jedes Ereignis wird separat geplant und besteht aus Aktionen, die weitere Ereignisse planen. Charakteristisch für die ereignisorientierte Modellierung ist, dass Ereignisse, die zusammenhängen, nicht zusammenhängend beschrieben werden. Dadurch kann eine ereignisorientierte Modellbeschreibung unübersichtlich und schwer nachvollziehbar sein.

Ereignisse betreffen häufig ein bestimmtes Systemelement und sind durch dieses Element einem Ablauf zugehörig, der das Systemelement betrifft. Die prozessorientierte Modellierung erlaubt eine kompakte und zusammenhängende Beschreibung dieser Ereignisse. Eine Prozessbeschreibung besteht aus Sprachelementen, die (a) auf den Eintritt eines Ereignisses warten, (b) neue Ereignisse planen, (c) Ereignisse auslösen und (d) nicht unmittelbar Ereignis-relevante Zustandsänderungen ausführen. Die

Prozessbeschreibung gibt dabei eine oder mehrere mögliche Reihenfolgen für Abarbeitung dieser Sprachelemente vor. Je nach Sprache, können Sprachelemente zu einer oder mehreren dieser Kategorien zugeordnet werden.

Die aktivitätsorientierte Sicht bezeichnet Abläufe in Sprachen, deren Modelle aus Aktivitäten zusammengesetzt sind und die keine allgemeinen Abstraktionskonzepte in Form von Objekten und Funktionen besitzen. Zu diesen Sprachen gehört GPSS. Eine Aktivität besteht hier aus Aktionen, die in einer vordefinierten Reihenfolge abgearbeitet werden. Der einzige Zustand einer Aktivität ist die nächste auszuführende Aktion. Die Aktionen selbst können dabei zustandsbehaftet sein und Werte aufnehmen oder die weitere Ausführung blockieren. In GPSS werden diese Aktionen als Blöcke bezeichnet. Die aktivitätsorientierte kann als Vereinfachung der prozessorientierten Sicht gesehen werden. Es gibt keinen Objektbezug und eine Aktivität kann im Gegensatz zu einer Funktion nicht mehrfach mit verschiedenen Werten aufgerufen werden. Stattdessen kann die Aktivität nur mehrfach mit den gleichen Werten betreten werden.

Simulationssprachen können die Modellierung aus einer oder sogar aus mehreren Weltansichten erlauben. Die unterstützten Weltansichten und die Art ihrer Implementierung, ergeben viele mögliche Realisierungen von Simulationssprachen, die im Laufe der Zeit entstanden sind. Wie effizient Modelle in einer Sprache beschrieben werden können, hängt im Allgemeinen vom untersuchten System und von der Fragestellung ab.

Prozessorientierte Beschreibungen haben eine hohe Verständlichkeit und werden deshalb von den meisten Simulationssprachen unterstützt. Die vorliegende Arbeit setzt den Schwerpunkt deshalb auf prozessorientierte Beschreibungen.

2.2.4 Anforderungen diskreter Simulationssprachen

Nach Nance [46] unterstützen alle diskreten Simulationssprachen sechs grundlegende Anforderungen:

1. die Erzeugung von Zufallszahlen, um Unsicherheiten mit Hilfe eines stochastischen Teilmodells auszudrücken,
2. die Transformation von Zufallszahlen zu statistischen Verteilungen,
3. Verarbeitungsmechanismen mit denen Objekte erzeugt, gelöscht, verändert und zu Listen hinzugefügt und entfernt werden können,
4. statistische Analysemöglichkeiten für die Bewertung von Modellgrößen,
5. Mechanismen zur Berichterzeugung, die den Entscheidungsprozess in der Bewertung von Simulationsläufen unterstützen und
6. einen Mechanismus, der die Repräsentation der Modellzeit und ihren Fortschritt erlaubt.

Die Sprachen unterscheiden sich jedoch in dem Maße in dem sie die Anforderungen erfüllen und in der Art und Weise wie sie erfüllt werden. Dies hat Auswirkungen auf die Effizienz der mit ihnen durchgeführten Simulationen.

2.2.5 Original, Modell und Simulationssprache

Abb. 2.1 zeigt den Zusammenhang zwischen verschiedenen Begriffen im Originalsystem, im Modell und in der Simulationssprache, um so Klarheit bei der Verwendung dieser Begriffe zu schaffen.

In einem Originalsystem, das als Vorlage für ein Modell dient, bilden die Elemente des Systems eine Struktur. Die Struktur unterliegt Veränderungen durch Prozesse, die sich zeitlich und räumlich überlagern können und zu Zustandsänderungen in der Struktur des Systems führen.

In einem Modell wird eine Menge ähnlicher Prozesse, die bestimmte Eigenschaften und ein bestimmtes Verhalten gemeinsam haben, als Prozessdefinition beschrieben. Eine Prozessdefinition ist eine Schablone, die als Vorlage für ähnliche Prozesse dient. Ein Prozess bezeichnet im Modell eine Entität, die erst während der Ausführung des Modells existiert. Jede Prozessdefinition enthält eine Verhaltensbeschreibung. Die Prozesse werden entsprechend der Verhaltensbeschreibung parallel in der Modellzeit ausgeführt. Systemelemente, die kein eigenes Verhalten besitzen, werden durch einfache Strukturobjekte dargestellt, die einer Strukturdefinition folgen.

In einer prozessorientierten Simulationssprache erfolgt eine Strukturdefinition mit einer passiven Klasse, die Attribute und Methoden enthält. Eine aktive Klasse entspricht einer Prozessdefinition und besitzt eine ausgezeichnete Methode, die das Verhalten der aktiven Objekte der Klasse beschreibt. In der Sprache DBL ist das Verhalten in einem actions-Teil festgelegt. Eine aktive Klasse kann die gleichen Strukturen wie eine passive Klasse definieren. Ein aktives Objekt ist eine Instanz einer aktiven Klasse, die während der Ausführung des Modells existiert. Das aktive Objekt hat Werte für die Attribute und besitzt eine aktuelle Position der Ausführung seines Verhaltens. Außerdem kann das aktive Objekte Methoden aufrufen und verwaltet in diesem Fall eine Call-Stack-Datenstruktur für die aufgerufenen Funktionen.

2.2.6 Historisch bedeutsame Sprachen

Die erste Sprache

Die erste erfolgreiche zeitdiskrete Simulationssprache ist das Generale Purpose Simulation System (GPSS) [30], das im Jahre 1961 von Geoffrey Gordon entwickelt wurde. Die damalige Bezeichnung der Sprache als Simulationssystem unterstreicht die Anforderungen zur Durchführung und Auswertung von Simulationen, die über die reinen Konzepte einer Sprache hinausgehen. Heute umfasst die Bezeichnung eines solchen Systems als Sprache auch diejenigen Systeme, die zur Benutzung einer Sprache notwendig sind.

GPSS ist eine prozessorientierte konventionelle Sprache. Sie orientiert sich mit ihren Konzepten an den erfolgreichen Programmiersprachen der damaligen Zeit, die heute als Assembler-Sprachen bezeichnet werden. Ähnlich wie ein Programm in Assembler besteht ein Programm in GPSS aus Anweisungen, die in GPSS als Blöcke bezeichnet werden. Die Anweisungen werden nacheinander abgearbeitet, wobei auf Zustandsgrößen durch einfache Bezeichner Bezug genommen wird. Im Gegensatz zu einer Assembler-Sprache werden die Anweisungen jedoch unter Berücksichtigung einer Modellzeit ausgeführt. Damit bietet GPSS als erste Sprache eine Abstraktion für

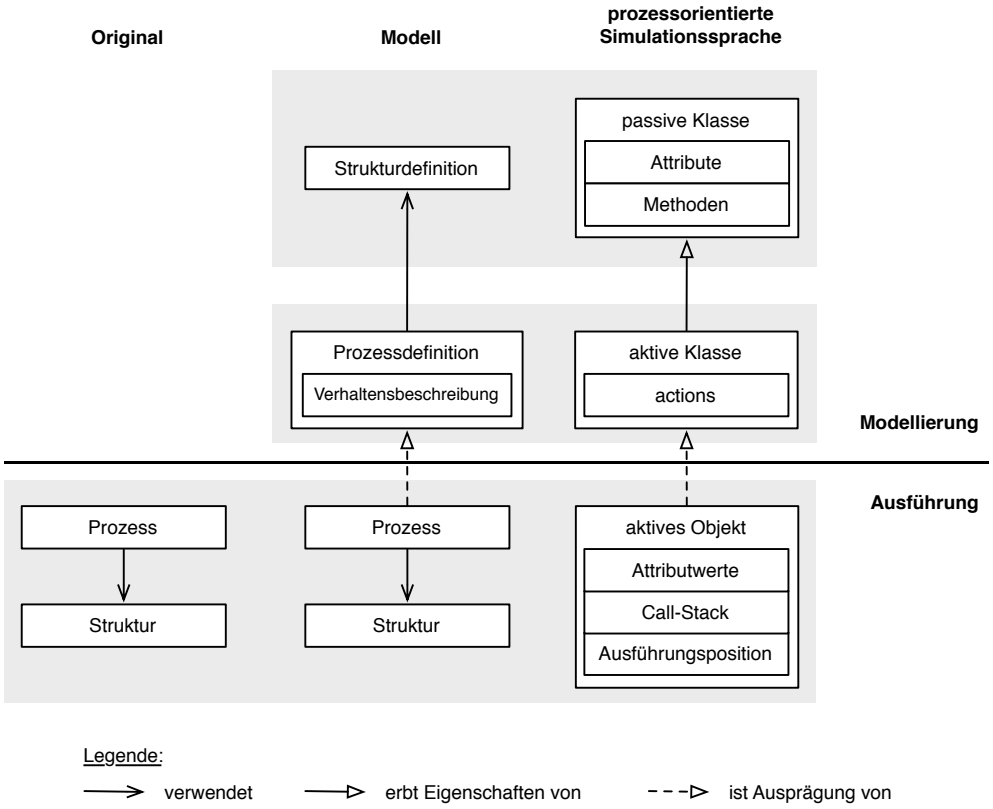

Abbildung 2.1: Zusammenhang zwischen verschiedenen Begriffen im Originalsystem, im Modell und in einer prozessorientierten Simulationssprache.

eines der zentralen Konzepte von Simulationssprachen: die quasi-parallele Ausführung mehrerer paralleler Abläufe in Abhängigkeit einer Modellzeit. Die technischen Mechanismen dieser Ausführung bleiben durch die Abstraktion in Form von Konzepten der Sprache vor dem Modellierer verborgen. Die Abstraktion ist somit leicht einsetzbar und das Modell bleibt verständlich.

Listing 2.1 zeigt ein einfaches Modell in GPSS, das aus einer Sequenz von vier Anweisungen besteht und die Verarbeitung von Einheiten mit Hilfe einer begrenzten Ressource nachbildet. Die GENERATE-Anweisung beschreibt das zeitlich versetzte Eintreffen von sogenannten Transaktionen, die Objektstrukturen darstellen, die durch GPSS vordefiniert sind. Die Anweisungen SEIZE und RELEASE beschreiben die Belegung und die Freigabe einer Ressource, die von höchstens einer Transaktion belegt werden kann. Hat eine Transaktion die Ressource belegt, so wird die durch die ADVANCE-Anweisung angegebene Modellzeit verbraucht. Transaktion werden durch GENERATE erzeugt und durchlaufen die einzelnen Anweisungen in der angegebenen Reihenfolge. Mit Ausführung der letzten Anweisung hören sie auf zu existieren. Bei einer Simulation werden am Ende Ergebnisgrößen zur Modellbewertung wie z.B. die Auslastung der Ressource und deren durchschnittliche zeitliche Belegung ausgegeben.

```
1 GENERATE 6,4
2 SEIZE M
3 ADVANCE 7,1
4 RELEASE M
```

Listing 2.1: Einfaches Modell in GPSS.

Das Modell in Listing 2.1 enthält zwei Arten von Zeitereignissen sowie ein Zustandsereignis. Die Anweisungen GENERATE und ADVANCE erzeugen zukünftige Zeitereignisse zu denen eine bestimmte Transaktion von einer Anweisung zur nächsten Anweisung wechselt. Die Zeitangaben sind dabei relativ zur aktuellen Modellzeit. Die Anweisungen SEIZE und RELEASE gehören dagegen zu einem Zustandsereignis. Betritt eine Transaktion die SEIZE-Anweisung, so ändert diese den Zustand der Ressource M auf belegt. Andere Transaktionen warten nun bis die Ressource wieder frei wird. Das Freiwerden ist ein Zustandsereignis, das erst durch das Betreten der RELEASE-Anweisung ausgelöst wird.

Obwohl Ereignisse, die einen bestimmten Prozess betreffen, in GPSS zusammenhängend beschrieben werden, wird GPSS als transaktionsorientierte Modellierungssprache bezeichnet. Eine Transaktion ist dabei ein Systemelement, das sich durch das System bewegt oder bewegt wird. Es ist kein stationäres Systemelement wie z.B. eine Maschine. GPSS ist dazu geeignet, Transaktionen als Prozesse zu beschreiben. Das System wird dabei aus Sicht dieser sich bewegenden Systemelemente modelliert. GPSS ist dagegen nicht geeignet, ein System aus Sicht der stationären Systemelemente zu beschreiben, da diese bereits als spezielle Anweisungen in die Sprache integriert sind und nicht verändert werden können. Deshalb bezeichnet man GPSS als transaktionsorientiert. Eine Sprache ist dagegen prozessorientiert, wenn sie auch die Beschreibung der Prozesse von stationären Systemelementen unterstützt.

Objektorientierung

Bereits kurze Zeit später, im Jahre 1966, gelang Ole-Johan Dahl und Kristen Nygaard mit der Programmiersprache Simula die Einführung der objektorientierten Modellierung, die heute ein fester Bestandteil vieler anderer Programmiersprachen ist. Das Klasse-Objekt-Konzept ermöglicht eine struktur- und verhaltensäquivalente Abbildung der Elemente eines Systems durch Objekte mit Attributen, Objektreferenzen und Methoden. Es ist damit ein wichtiger Schritt in Richtung verständlicher Simulationsmodelle, da die Objekte im Modell einen direkten Bezug zu den Elementen des modellierten Systems besitzen.

Die Sprache Simula enthält allgemeine Konzepte, die für die Implementierung von Algorithmen geeignet sind und nicht spezifisch für Simulationsmodelle vorgesehen sind. Die Sprache führt Simulationskonzepte dagegen über eine Standardbibliothek ein, die nur eine Schnittstelle für die Konzepte festlegt. Die Standardbibliothek muss neben einem Compiler für die Sprache ebenfalls implementiert werden. Da die Sprache nur inklusive einer Implementierung der Standardbibliothek als vollständig angesehen werden kann, ist sie ebenfalls den konventionellen prozessorientierten Simulationssprachen zugehörig.

Die objektorientierten Konzepten machten es in Simula, im Gegensatz zu GPSS, möglich, die für Simulationsmodelle in unterschiedlichen Domänen hilfreichen Elemente nicht in die Sprache selbst zu integrieren, sondern sie mit der Sprache auszudrücken und in einer Programmbibliothek zur Verfügung zu stellen. Dazu gehören z.B. Elemente für die Modellierung von Warteschlangen und Ressourcen. In Simula werden diese durch die Programmbibliothek DEMOS [9] in Form von Klassen definiert. Die Sprache kann damit auf einen überschaubaren Teil von Kernkonzepten fokussiert werden. Zudem können weitere Elemente einfacher zur Bibliothek als zur Sprache selbst hinzugefügt werden. Das Klasse-Objekt-Konzept erlaubt in seiner flexiblen Anwendbarkeit so auch eine einfache Erweiterung der Sprache. Die Syntax einer Erweiterung ist dabei jedoch fest durch die Sprache, in diesem Fall Simula, vorgegeben.

Viele Konzepte von Simula und in Simula entwickelter Bibliotheken wurden mit der Zeit in andere Sprachen übernommen, die heute eine sehr viel stärkere Verbreitung aufweisen. Zu diesen Sprachen gehört C++, die mit ihrer laufzeiteffizienten Ausführbarkeit besonders gut für die Simulation geeignet ist. Ähnlich wie DEMOS für Simula, gibt es für C++ die Programmbibliothek ODEMx [28], die am Lehrstuhl Systemanalyse entwickelt wird, und die Modellierungskonzepte für parallele Prozesse, sowie deren Interaktion und Ressourcenbedarf bereitstellt. ODEMx macht sich dabei die Möglichkeiten von C++ für die laufzeiteffiziente Implementierung quasi-paralleler Prozesse zu nutze.

Ein ähnlicher Vertreter wie ODEMx für C++ ist in Java die Programmbibliothek DESMO-J [44]. Sie stellt im zeitdiskreten Bereich die gleichen Konzepte wie ODEMx in der Sprache Java bereit. Der Vorteil von Java ist eine große Entwicklergemeinschaft, durch die viele nützliche Programmbibliotheken vorhanden sind. Der Nachteil von Java sind jedoch die weniger laufzeiteffizienten Möglichkeiten für die Implementierung quasi-paralleler Prozesse.

Grafische Sprachen

Neben textuellen Simulationssprachen entstanden im Laufe der Zeit auch grafische Sprachen, eingebettet in interaktive Oberflächen. Heute bekannte Vertreter sind Simulink [2] und Plant Simulation [6].

Plant Simulation ist eine grafische Simulationssprache für das Anwendungsgebiet Materialflusssimulation in Fabriken. Die Sprache besteht aus grafischen Blöcken, für z.B. Maschinen und Warteschlangen, die durch Flusskanten verbunden sind und so einen Materialflussgraphen abbilden. Zusätzlich kann die Struktur von Materialobjekten definiert werden, die entsprechend des Graphen von Block zu Block weiterbewegt werden. Diese Konzepte haben ihren Ursprung in der Sprache GPSS. Auch hier gibt es bereits Blöcke für Maschinen sowie Transaktionen für die Materialflussobjekte. Wie in GPSS, wird auch in Plant Simulation der Materialfluss aus der Sicht der Materialobjekte beschrieben. Im Gegensatz zu GPSS ist es jedoch möglich, das Verhalten der Blöcke imperativ zu beschreiben. Auf diese Weise kann Plant Simulation durch neue Blöcke, die für eine spezielle Fertigungsanlage hilfreich sein können, erweitert werden.

2.3 Domänenspezifische Sprachen

Im Gegensatz zu Programmiersprachen und allgemeinen Simulationssprachen, gibt es Sprachen, die spezielle Konzepte für eine bestimmte Domäne bereitstellen, und als domänenspezifische Sprachen (DSLs) bezeichnet werden. Ein Modell, das in einer DSL beschrieben wird, ist prägnant, da die DSL passgenaue Konzepte für die Modellierung bereitstellt. Eine Definition für eine DSL wird von Ghosh [29] in seinem Buch *DSLs in Action* gegeben:

> A DSL is a programming language that's targeted at a specific problem [...] It contains the syntax and semantics that model concepts at the same level of abstraction that the problem domain offers.

Eine DSL zielt also immer auf ein spezielles Problemgebiet ab. Sie enthält eine Syntax und Semantik, welche die Konzepte auf dem gleichen Abstraktionslevel wie die Problemdomäne nachbilden.

Eine DSL ermöglicht die Verknüpfung von Abstraktionen des Problemgebiets in der gleichen Art und Weise auch im Lösungsraum. Der Lösungsraum entspricht dabei der Menge an möglichen Sprachinstanzen, die mit einer DSL formulierbar sind. Der Lösungsraum stellt also ähnliche Konzepte bereit, wie man sie für das Problemgebiet benötigt. Verwendet man dagegen eine Mehrzwecksprache im Lösungsraum, so weichen deren Konzepte, aufgrund ihrer allgemeinen Anwendbarkeit, in einem bestimmten Umfang von den Konzepten des Problemgebietes ab. Es ist davon auszugehen, dass die entsprechenden Sprachinstanzen schwieriger nachvollziehbar und umfangreicher sind als die mit einer DSL formulierten.

Gosh beschreibt auch, dass eine DSL, im Unterschied zu einer Mehrzwecksprache, intuitiver und leichter anwendbar für Benutzer aus dem Problemgebiet ist. Es wird davon ausgegangen, dass diese meistens wenig Programmiererfahrung besitzen und mit den vollen Möglichkeiten einer objektorientierten Programmiersprache überfordert sind. Durch das hohe Abstraktionsniveau einer DSL, kann sich ein Domänenexperte

auf die eigentliche Problemlösung konzentrieren, und muss sich nicht mit Details der Programmierung auseinandersetzen. Eine DSL bietet dazu ein begrenztes und passendes Vokabular. Eine DSL wird als Gegenstück zu einer Mehrzwecksprache gesehen. Beispiele für Mehrzwecksprachen sind Programmiersprachen, aber auch Simulationssprachen, wenn diese nicht für ein spezielles System konzipiert sind, sondern sich zur Modellierung unterschiedlicher Klassen von Systemen einsetzen lassen.

Aus dieser Beschreibung wird bereits ersichtlich, dass es keine klare Abgrenzung von DSLs zu anderen Computersprachen gibt. Diese unklare Abgrenzung beschreibt auch Fowler [25] in seinem Buch *Domain-Specific Languages*:

> „Domain-specific language“ is a useful term and concept, but one that has very blurred boundaries. Some things are clearly DSLs, but others can be argued one way or the other. [...]

Für Fowler ist das Schlüsselmerkmal einer DSL, eine *eingeschränkte Ausdruckskraft* gegenüber einer Programmiersprache. Das bedeutet, dass eine DSL bestimmte Sprachelemente einer Programmiersprache bewusst nicht enthält. Ein Beispiel für eine DSL ist SQL. Die Sprache SQL dient der Definition von Datenstrukturen in relationalen Datenbanken. Ursprünglich enthielt SQL keine Variablen und auch keine Prozeduren und Anweisungen. Damit ist eine *eingeschränkte Ausdruckskraft* gegeben. Es gibt jedoch auch eine Variante von SQL mit dem Namen PL/SQL (Procedural Language/Structured Query Language) [69], die diese Sprachelemente enthält und im Gegensatz zu SQL Turing-vollständig ist. PL/SQL besitzt also keine *eingeschränkte Ausdruckskraft* und ist nach Fowler keine DSL.

Es gibt weitere Sprachen, die zwar in einer bestimmten Domäne entstanden sind und entsprechende Domänenkonzepte enthalten, jedoch auch die Ausdruckskraft einer Programmiersprache besitzen. Fowler gibt hier als Beispiel die Sprache XSLT an, eine Sprache zur Transformation von XML-Dokumenten, die jedoch auch für die Beschreibung eines allgemeinen Algorithmus eingesetzt werden kann.

Im Bereich der Simulation, gibt es mit SLX eine Sprache, die ähnlich wie Simula, zugleich eine objektorientierte Programmiersprache und auch eine Simulationssprache ist. SLX eignet sich aufgrund integrierter Sprachelemente, wie z.B. einer Anweisung für den Modellzeitverbrauch, gut zur Formulierung von Simulationsmodellen. Andererseits enthält SLX jedoch auch Sprachelemente von Programmiersprachen. Man könnte SLX auch verwenden, um ein Problem zu lösen, das losgelöst ist von einer Simulation, z.B. der Implementierung eines Sortieralgorithmus. Nach Fowler ist SLX keine DSL.

Nach Fowler ist ein weiteres Unterscheidungsmerkmal notwendig, um eine Sprache als DSL zu bezeichnen. Dieses Merkmal ist *die Art und Weise* wie eine Sprache benutzt wird. Die Bezeichnung einer Sprache als DSL ist damit abhängig von dem konkreten Problem für dessen Lösung sie eingesetzt wird.

Die Zuordnung einer Sprache zur Klasse der DSLs oder der Mehrzwecksprachen ist relativ zur Problemsituation zu betrachten, die mit einer Sprachinstanz gelöst werden soll. Eine Programmiersprache ist immer eine Mehrzwecksprache und niemals eine DSL. Eine Simulationssprache ist in Bezug auf das Anwendungsgebiet Simulation eine DSL. Eine Simulationssprache ist in Bezug auf das Anwendungsgebiet der Produktionssysteme eine Mehrzwecksprache wenn sie Sprachelemente enthält, die auch zur

Modellierung anderer Systemklassen geeignet sind. Eine Simulationssprache für Produktionssysteme wäre in diesem Fall eine DSL. Wenn jedoch ein spezielles Produktionssystem betrachtet wird, so kann die Simulationssprache für Produktionssysteme wieder zu einer Mehrzwecksprache werden. Dies ist abhängig davon wie geeignet die Sprachelemente für den speziellen Fall noch sind. Die Bezeichnung hängt also immer vom konkreten Problemraum ab. Wird SLX also für die Simulationsmodellierung eingesetzt, dann ist SLX eine DSL für die Domäne Simulation. Setzt man SLX jedoch für die Beschreibung eines Sortieralgorithmus ein, so ist SLX eine Mehrzwecksprache.

Diese Überlegungen führen mich zur Definition von Sprache und DSL.

Definition 2.1 (Sprache)**.** Eine Sprache enthält mindestens zwei Arten von Elementen, die nach bestimmten Regeln frei kombinierbar sind. Die möglichen Kombinationen sind nicht endlich aufzählbar. Durch eine konkrete Kombination wird eine Lösung für ein konkretes Problem formuliert.

Angenommen eine Sprache für einfache Zustandsautomaten ist durch die Sprachelemente Zustandsmaschine, Zustand und Transition definiert. Dann bilden diese Sprachelemente zusammen eine Sprache, da Beziehungen zwischen den Sprachelementen zur Lösung eines Problems hergestellt werden müssen. Jedes Sprachelemente für sich ist dagegen keine Sprache, da sie isoliert nicht zur Beschreibung einer Lösung ausreichend sind. Angenommen eine Sprache enthält eine spezielle Anweisung zur Beschreibung einer Endlosschleife, dann ist das entsprechende Sprachelement keine Sprache, da es keine Beziehungen zu weiteren Sprachelementen gibt.

Definition 2.2 (Domänenspezifische Sprache (DSL))**.** Eine domänenspezifische Sprache (DSL) ist eine Computersprache, die Sprachelemente enthält, mit denen sich Lösungen für eine bestimmte Klasse von Problemen in einem begrenzten Problemgebiet prägnanter und nachvollziehbarer beschreiben lassen als dies mit einer Programmiersprache möglich ist. Eine DSL enthält dabei Sprachelemente, die für eine andere Problemklasse ungeeignet sind. Eine Verwendung in einer anderen Problemklasse führt zu Beschreibungen, die schwer nachvollziehbar und nicht mehr prägnant formulierbar sind. Eine DSL definiert eine eigene Syntax, die nicht mit der Syntax einer gewählten Implementierungssprache übereinstimmt und für die ein separater Parser notwendig ist.

2.3.1 Klassifikation von DSLs

In der Literatur zu DSLs, hat sich eine Klassifikation von DSLs nach der Art ihrer Implementation etabliert. Diese Klassifikation wurde ursprünglich von Fowler eingeführt. Er unterscheidet im Wesentlichen externe DSLs und interne DSLs. Zusätzlich unterscheidet er als dritte Klasse Sprachumgebungen (language workbenches). Ich werde zunächst erklären, was die wesentlichen Unterscheidungsmerkmale dieser Klassen sind. Danach setze ich mich kritisch mit dieser Klassifikation auseinander.

Externe DSLs

Ein externe DSL besitzt eine eigenständige Syntax und Semantik, die speziell für die DSL entwickelt wurde und nicht aus einer anderen Sprache integriert wurde. Aus

diesem Grund benötigt eine externe DSL auch vollkommen neue Sprachwerkzeuge. Dazu gehört in jedem Fall ein Werkzeug, das Sprachinstanzen in eine weiterverarbeitbare Darstellung überführt, sowie ein Programm, dass die Semantik von Sprachinstanzen realisiert. Mögliche Sprachwerkzeuge sind also ein Parser, ein Interpreter und ein Editor.

Eine externe DSL kann so konzipiert werden, dass sie ideal geeignet ist, um Lösungen für bestimmte Probleme zu formulieren. Dabei ist die Ausdruckskraft gegenüber Mehrzwecksprachen eingeschränkt. Beispiele für externe DSLs sind SQL, Ant, grep und BNF. Diese Sprachen sind für spezielle Probleme konzipiert. Sie können nicht verwendet werden, um ein beliebiges Problem mit einem Algorithmus zu beschreiben. Diese Sprachen sind damit nicht Turing-vollständig. Eine nicht vorhandene Turing-Vollständigkeit ist jedoch keine hinreichende Bedingung für eine externe DSL. So ist die Sprache SDL zur Beschreibung reaktiver Systeme Turing-vollständig.

Die Werkzeuge einer externen DSL müssen nicht in jedem Fall komplett neu in einer Programmiersprache implementiert werden. Es ist möglich, Teilaspekte einer Sprache in einer Spezialsprache zu beschreiben und ein Werkzeug aus einer solchen Beschreibung automatisch zu generieren. Dazu gehören z.B. Parser, die aus der Beschreibung einer Grammatik in BNF oder EBNF automatisch generiert werden. Eine neuere Entwicklung ist die Möglichkeit auch textuelle Editoren mit Eingabeunterstützung und Syntaxhervorhebung aus einer Sprachbeschreibung zu erzeugen. Sprachentwicklungswerkzeuge, die eine solche Generierung unterstützen, sind TEF [62] und Xtext [36].

In jedem Fall, muss der Sprachentwickler einer externen DSL eine separate Beschreibung erstellen, bevor ein Sprachwerkzeug verfügbar ist. Der Vorteil einer externen DSL liegt in den weitreichenden Möglichkeiten zur Definition einer geeigneten Syntax und Semantik. Ihr Nachteil liegt jedoch in den entstehenden Kosten für die Entwicklung von Sprachwerkzeugen. Werkzeuge zur Sprachentwicklung, die es erlauben Teilaspekte von Sprachen in Spezialsprachen zu beschreiben, verringern diese Kosten jedoch. Zu diesen Werkzeugen gehören auch Sprachumgebungen.

Sprachumgebungen

Eine Sprachumgebung (language workbench) erlaubt es, mehr als einen Teilaspekt einer Sprache in einer integrierten Entwicklungsumgebung zu beschreiben und Sprachwerkzeuge für die beschriebene Sprache automatisch zu generieren. Eine Sprachumgebung lässt sich zur Entwicklung einer externen DSL einsetzen. Eine externe DSL muss jedoch nicht mit einer Sprachumgebung entwickelt werden. Sprachumgebungen sind also eine Teilmenge externer DSLs.

Beispiele für Sprachumgebungen sind Xtext, MPS und Spoofax. Sie erlauben neben der Beschreibung einer textuellen Syntax, auch die Definition weiterer Aspekte wie z.B. die Beschreibung eines Typsystems oder die Beschreibung von Gültigskeitsbereichen für Sprachelemente sind mit diesen Umgebungen möglich. Zu den Sprachentwicklungswerkzeugen, die keine Sprachumgebungen sind, gehören z.B. TEF, ANTLR [53] und EProvide [58], da diese sich nur auf einen Sprachaspekt konzentrieren. Bei TEF wird ein textueller Editor erzeugt, bei ANTLR ein Parser und bei EProvide ein Interpreter.

Interne DSLs

Eine interne DSL ist immer eingebettet in eine bestehende Sprache, die häufig als Hostsprache bezeichnet wird. Die interne DSL verwendet dabei die Hostsprache in einer bestimmten Art und Weise, so dass der Eindruck einer neuen Sprache entsteht, obwohl letztendlich nur die Sprachelemente der Hostsprache verwendet werden. Je nach Hostsprache unterscheiden sich dabei die Möglichkeiten mit denen einer DSL eine bestimmte Syntax gegeben werden kann.

Bei einer internen DSL können die Sprachwerkzeuge der Hostsprache auch für Sprachinstanzen der DSL benutzt werden. Die Werkzeuge müssen also nicht neu entwickelt werden. Ein Nachteil ist dabei, dass die Werkzeuge der Hostsprache nicht für die speziellen Sprachelemente der DSL ausgelegt sind. Damit sind Sprachinstanzen in der DSL nur auf dem Level der Hostsprache zugänglich.

Das Schlüsselmerkmal der eingeschränkten Ausdruckskraft, trifft auf eine interne DSL nicht zu, da der gesamte Umfang der Hostsprache zur Verfügung steht. Nach Fowler bezieht sich die eingeschränkte Ausdruckskraft bei einer internen DSL auf die Art und Weise der Benutzung. Bei der Verwendung der DSL setzt man dabei nur eine Teilmenge der Elemente der Hostsprache ein.

Eine Bibliothek ist ebenfalls ein Konzept, bei dem für Lösungen in einem bestimmten Problemgebiet auf die Sprachelemente einer Hostsprache zurückgegriffen wird. Die Bibliothek stellt Klassen, Operationen und Variablen bereit, die Bezeichnungen aus dem Vokabular der Domäne tragen. Auf diese Weise wird die Formulierung von Lösungen erleichtert.

Es gibt dabei keine klare Unterscheidung zwischen einer Bibliothek und einer internen DSL. Fowler argumentiert, dass der Unterschied in der Natur der Sprache liegt. Eine Bibliothek definiert nur ein Vokabular, während eine interne DSL eine Grammatik definiert. Fowler spricht zwar explizit von Grammatik, er meint damit aber keine Grammatik als Menge von Regeln, die eine konkrete Syntax beschreiben, sondern eine Beschreibung, die mehr Freiheiten in der Syntaxdefinition erlaubt, als es die klassische Aneinanderreihung von Operationsaufrufen in einer objektorientierten Sprache erlaubt. Eine interne DSL vermittelt dabei das Gefühl zusammenhängende Sätze oder Aussagen zu formulieren, während eine Bibliothek lediglich eine Aneinanderreihung von Kommandos erlaubt. Entscheidend ist für ihn, ob die Darstellung, die eine Sprache für einen bestimmten Systemaspekt erlaubt, auch der Art und Weise eines Menschen entspricht mit diesem Systemaspekt zu interagieren. In diesem Fall ist die Sprache für ihn eine interne DSL. Es gibt also keine klare Definition dafür, was interne DSL ist und bei welchen Hostsprachen man von interner DSL oder Bibliothek spricht.

Aufgrund der nicht klaren Definition des Begriffs interne DSL, werden gewöhnliche Bibliotheken in einigen Sprachen heute oft als DSLs bezeichnet. Man fragt sich dann zu Recht, ob diese Bibliotheken tatsächlich DSLs, also eigene Sprachen sind. Dieses Phänomen ist z.B. in der Sprache Ruby und ihrer Gemeinschaft vorhanden.

Fowler und Gosh führen Ruby on Rails (auch kurz als Rails bezeichnet) als Beispiel für eine interne DSL an. Die Dokumentation von Rails [8] beschreibt Rails jedoch unter dem Punkt „What is Rails?“ als Framework für die einfache Entwicklung von Web-Anwendungen, das in Ruby geschrieben ist. Die Bezeichnung DSL findet sich in

der insgesamt 11 Kapitel umfassenden Dokumentation lediglich ein einziges Mal. Im Abschnitt 4.3 befindet sich die folgende Beschreibung:

> This is your application's routing file which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions.

Diese Beschreibung bezieht sich auf die Ruby-Datei in Listing 2.2.

```
1 Rails.application.routes.draw do
2   get 'welcome/index'
3 end
```

Listing 2.2: Konfigurationsdatei in Ruby Rails.

Einträge in dieser Datei sind Teil der Konfiguration. Sie beschreiben, wie eingehende Anfragen an Controller und Actions von Rails weitergeleitet werden. Rails sieht sich selbst also vordergründig nicht als DSL. Dennoch wird von anderen die Auffassung vertreten, Rails sei eine DSL. Betrachtet man die Konfigurationsdatei, erkennt man zunächst keine Funktionen, Variablen oder andere Konstrukte aus Mehrzwecksprachen. Dieses hängt jedoch nur mit der besonders flexiblen Syntax von Ruby zusammen. Die Syntax erlaubt das Weglassen von Klammern, sofern die Zuordnung der Argumente zur Funktion eindeutig bleibt. Außerdem gibt es in Ruby Lambda-Ausdrücke mit denen anonyme Funktionen als Argumente an eine Funktion übergeben werden können. Wegen der besonderen Syntax sind diese Konstrukte einer höheren Programmiersprache jedoch nicht direkt erkennbar. Listing 2.2 könnte man auch wie in Listing 2.3 dargestellt notieren:

```
1 Rails.application.routes.draw() {
2   get('welcome/index')
3 }
```

Listing 2.3: Konfigurationsdatei in Ruby Rails mit eingesetzten Klammern.

Man erkennt nun einen Aufruf der Funktion draw, der als Argument eine anonyme Funktion übergeben wird, die wiederum eine Funktion get aufruft. Tatsächlich enthält die Konfigurationsdatei also eine Sequenz von Funktionsaufrufen. Der Eindruck einer Sprache entsteht aufgrund fehlender einfacher und geschweifter Klammern.

Ähnlich verhält es sich mit der Ruby-Bibliothek Rake [84]. Diese dient der Beschreibung von sogenannten Build-Prozessen, die Vorgänge zur automatischen Erzeugungen von Anwendungsprogrammen aus vielen Einzelschritten beschreiben.

> Rake is a Make-like program implemented in Ruby. Tasks and dependencies are specified in standard Ruby syntax.

Eine Beschreibung in Rake besteht aus Aufgaben (tasks) und Abhängigkeiten. Im Gegensatz zu Make, besitzt Rake keine eigene Syntax, sondern verwendet die Syntax von Ruby. Eine einfache Beschreibung in Rake, die eine Aufgabe test festlegt, und von zwei Aufgaben compile und dataLoad abhängt, ist in Listig 2.4 dargestellt.

```
task :test => [:compile, :dataLoad] do
  # run the tests
end
```

Listing 2.4: Task in Ruby Rake.

Fügt man die weggelassenen Klammern hinzu, ergibt sich das in Listing 2.5 abgebildete Programm.

```
task({:test => [:compile, :dataLoad]}) {
  # run the tests
}
```

Listing 2.5: Task in Ruby Rake mit eingesetzten Klammern.

task ist eine Funktion mit einem Parameter. Als Argument wird ein assoziatives Array übergeben, das unter dem Index :test als Wert ein normales Array mit den Werten :compile und :dataLoad enthält. Namen, die mit einem Doppelpunkt beginnen, werden in Ruby als Symbole bezeichnet. Ein Symbol ist immer eindeutig über seinen Namen identifiziert. Wird der gleiche Name also an verschiedenen Stellen verwendet, so verweist dieser immer auf das gleiche Symbol. Im Gegensatz dazu verweisen zwei Zeichenketten mit dem gleichen Inhalt nicht immer auf das selbe Objekt, das die Zeichenkette enthält.

In der Dokumentation von Rake wird der Begriff DSL nicht verwendet. Die Funktion task ist jedoch in einem Ruby-Modul mit der Bezeichnung DSL hinterlegt. Fowler gibt Rake ebenfalls als Beispiel für eine interne DSL an [26].

Rails und Rake sind Beispiele für Programmbibliotheken, die es erlauben bestimmte Konstrukte eines klassischen Programms durch eine flexible Syntax zu verbergen. Damit entsteht der Eindruck einer eigenen fokussierten Sprache, obwohl die Beschreibungen immer in der Programmiersprache Ruby erfolgen und immer Sprachelemente von Ruby enthalten. Ein Anwender einer internen DSL muss die Werkzeuge der Hostsprache verwenden, die nicht speziell für die DSL angepasst sind. In Bezug auf die Anwendung ist dies ein Nachteil, da der Anwender die Sprachelemente der Hostsprache auf Konzepte der DSL immer wieder selbst mental übertragen muss.

Zusammenfassend gelange ich daher zu der Erkenntnis, dass viele interne DSLs keine DSLs im Sinne einer eingeschränkten und fokussierten Sprache sind. Sie besitzen mehr Gemeinsamkeiten mit klassischen Programmbibliotheken, da die Syntax der Sprachelemente nicht frei definiert werden kann.

2.3.2 Einordnung von DMX zur Entwicklung von externen und internen DSLs

Mit dem metamodellbasierten Ansatz können externe DSLs entwickelt werden. Diese DSLs besitzen eine konkrete Syntax, die unabhängig von anderen Sprachen festgelegt ist. Sie bieten damit die größtmögliche Freiheit in der Definition einer neuen Sprache. Diese Freiheit erhöht jedoch auch den Aufwand für ihre Entwicklung. Die eingesetzten Metasprachen und die erzeugten Sprachwerkzeuge müssen im jeweiligen Fall erst integriert werden, so dass eine konsistente Modellierung in einer DSL möglich ist. Obwohl die objektorientierten Konzepte von Metamodellen die Wiederverwendung

abstrakter Sprachkonzepte in einer DSL bereits erlauben, ist dies für andere Sprachaspekte nicht möglich, da jeder Aspekt in einer eigenen unabhängigen Metasprache festgelegt ist.

Im Gegensatz zu einer externen DSL, ist die konkrete Syntax einer internen DSL durch eine sogenannte Host-Sprache, in die die DSL eingebettet ist, eingeschränkt. In einer internen DSL können die Konzepte der Host-Sprache direkt verwendet werden. Festlegungen in Bezug auf die Gültigkeit bestimmter Konzepte sind jedoch nicht möglich. Die Sprachwerkzeuge der Hostsprache können gleichermaßen für die DSL verwendet werden, stellen jedoch keine DSL-spezifischen Funktionen bereit.

Eine DSL, die als Spracherweiterung definiert ist, lässt sich jedoch nicht eindeutig einer dieser Klassen zuordnen. Die konkrete Syntax einer solchen DSL kann zwar nicht vollkommen unabhängig von der Basissprache, aber mit einer großen Freiheit ihr gegenüber, festgelegt werden. Zudem ist die DSL in die Basissprache eingebettet, da sie an genau festgelegten Stellen, weitere mögliche Ausprägungen von Basiselementen erlaubt. Die DSL kann außerdem bestehende Basiselemente wiederverwenden. Diese Art von DSL besitzt also Eigenschaften externer und interner DSLs. Da die Basissprache aber, genau wie eine Host-Sprache, die möglichen Sprachkonzepte einer DSL einschränkt, lassen sich DSLs, die durch Spracherweiterung definiert sind, eher den internen DSLs zuordnen.

2.3.3 Simulations-DSL

Da eine DSL im Allgemeinen nicht bereit steht und erst entwickelt werden muss, folgt daraus die Notwendigkeit effizienter Entwicklungen von DSLs. Im Besonderen muss diese Entwicklung im vorliegenden Fall auch effizient für Simulationssprachen sein, da diese spezifische Anforderungen besitzen, die berücksichtigt werden müssen. Ich bezeichne diese DSLs deshalb hier als Simulations-DSLs.

Eine Simulations-DSL muss ausreichende Ausdrucksmöglichkeiten für Sprachkonzepte zur Modellierung in verschiedenen Anwendungsbereichen bieten. Sie muss sich besonders effizient entwickeln lassen, da DSLs und insbesondere Simulationssprachen hochspezialisierte Sprache sind, die nur von wenigen Entwicklern betreut werden und damit nur geringe Ressourcen für die Implementierung von Tools bereitstehen. Außerdem muss eine Simulations-DSL eine hohe Laufzeiteffizienz in der Ausführung von Simulationen aufweisen, so dass viele Modellvarianten in kurzer Zeit untersucht werden können. Diese speziellen Anforderungen an Simulations-DSLs werden durch eine erweiterbare Simulationsbasissprache erfüllt.

Eine der zentralen Anforderungen an ein Simulationsmodell ist eine adäquate Beschreibung, die vorhersehbare Manipulationen des Modells ermöglicht. Objektorientierte Modelle stellen bereits einen wichtigen Schritt in diese Richtung dar. Dennoch unterliegen die Darstellungsmöglichkeiten den Syntaxvorgaben der eingesetzten objektorientierten Sprache. Eine domänenspezifische Sprachen erlaubt dagegen eine adäquatere Beschreibung, da die Syntax dem Problem entsprechend angepasst werden kann.

Eine Simulations-DSL enthält, im Gegensatz zu einer DSL, neben domänenspezifischen Sprachelementen auch Sprachelemente zur ereignis- oder prozessorientierten Modellierung. Diese können ein Teil der domänenspezifischen Sprachelemente sein.

Ein Beispiel hierfür sind Transitionen von Zustandsautomaten an denen eine Zeit angegeben ist. Eine solche Transition führt zu einem Zustandsübergang wenn ein Zeitereignis bei Ablauf der entsprechenden Zeit eintritt. Eine Zustandsautomatensprache, die in einem Simulationsmodell eingesetzt wird ist eine SDSL für die Domäne der reaktiven Systeme.

Produktionssysteme

Weitere Beispiele für SDSLs sind Simulationssysteme aus der Domäne der Produktionssysteme, die der Grundphilosophie von GPSS folgen. Dazu gehören Flexsim [43], PlantSimulation [6], ProModel[2], AutoMod[3] und WITNESS[4]. Diese Systeme bestehen primär aus einer grafischen Sprache zur Modellierung von Abläufen. Ein Modellierer kann dabei aus einer festgelegten Anzahl an grafischen Elementen auswählen, die eine bestimmte vordefinierte Funktion erfüllen. Diese werden über Flusskanten miteinander verbunden, die so den Weg von beweglichen Objekten festlegen. Die Eigenschaften von Objekten werden über Formulare angepasst. Wird ein neues statisches Objekt benötigt oder soll ein bestehendes Objekt in seinem Verhalten angepasst werden, kann häufig eine integrierte textuelle Programmiersprache verwendet werden. Im Beispiel von PlantSimulation wird hierfür eine Sprache mit dem Namen SimTalk bereitgestellt. Diese unterstützt eine prozessorientierte Modellierung des Verhaltens. Dabei kann auf Ereignisse gewartet werden und es können Zustandsinformationen anderer Objekte abgefragt werden.

Optische Nanostrukturen

Wider et al. [64] entwickelt eine DSL zur Beschreibung der Struktur photonischer Kristalle und der Experimente, die mit diesen durchgeführt werden. Photonische Kristalle sind spezielle optische Nanostrukturen, die von Physikern erforscht werden. Die DSL erlaubt eine adäquate Beschreibung sowohl der Strukturen als auch der Experimente, die mit Hilfe von Simulationen durchgeführt werden. Zur Analyse werden verschiedene Simulationsmethoden sowie Simulatoren eingesetzt, die sich hinsichtlich Genauigkeit und Ressourcenverbrauch unterscheiden. Durch die einheitliche Beschreibung auf der Basis einer DSL, können Beschreibungen für die verschiedenen Methoden und Simulatoren automatisiert abgeleitet werden. Damit ist die Beschreibung von ihrem konkreten Einsatz in einem Werkzeug entkoppelt und kann wiederverwendet werden.

Wider et al. setzt eine textuelle DSL ein, da eine Beschreibung in der DSL aus vielen mathematischen Ausdrücken besteht und sich eine textuelle DSL nach Diskussionen mit Domänenexperten als am besten geeignet herausgestellt hat. Er setzt zur Definition der DSL und der Werkzeuge das DSL-Entwicklungs-Framework Xtext [36] ein. Dabei muss das Konzept des Ausdrucks für die DSL erneut definiert werden, obwohl Ausdrücke in ihrer allgemeinen Form bereits eine gute Ausgangsbasis bieten würden. Häufig ist es sinnvoll, Ausdrücke für einfache mathematische Operationen wie z.B.

[2] http://www.promodel.com/
[3] http://www.automod.de
[4] http://www.lanner.com

eine Addition oder Multiplikation unter Berücksichtigung der entsprechenden Auswertungsreihenfolge bereits zur Verfügung zu haben. Des Weiteren kann das Konzept einer Funktion ebenfalls hilfreich sein. Dieses ist in der DSL jedoch nicht enthalten.

Aus der Arbeit von Wider et al. ergibt sich eine weitere Anforderung an Techniken zur DSL-Entwicklung. Häufig existieren in einer Domäne bereits Werkzeuge, die bei der Entwicklung einer DSL verwendet werden können. Eine Technik muss es gestatten, vorhandene Software zu integrieren. In der Arbeit von Wider et al. erfolgt diese Integration in der Beschreibung der Abbildung der DSL auf eine konkrete Simulationsmethode bzw. einen konkreten Simulator. Die Abbildungsziele sind dabei in Bezug auf die Zielsprache unterschiedlich. Für die eine Methode erfolgt eine Abbildung in eine werkzeugspezifische Skriptsprache. Für die andere Methode erfolgt die Abbildung dagegen in die funktionale Sprache Lisp. Die beiden Zielsprachen sind dabei sehr unterschiedlich. Die eingesetzte metamodellbasierte Sprachentwicklungstechnik ist für diese Problemstellung geeignet. Auf der Basis eines Metamodells können dabei in einer Modell-zu-Text- oder Modell-zu-Modell-Transformationssprache mehrere mögliche Abbildungsziele spezifiziert werden.

Modelle in der Nano-DSL sind nicht ereignisorientiert beschrieben. Die Ausführung erfolgt daher auch nicht mit einem ereignisorientierten Simulationsverfahren. Die Modelle enthalten in der Zielsprache Zustandsgrößen, die sich nach partiellen Differentialgleichungen zeitkontinuierlich ändern. Die Änderung erfolgt dabei durch Anwendung unterschiedlicher numerischer Lösungsverfahren. Eine Entwicklung der Nano-DSL mit Hilfe einer erweiterterbaren prozessorientierten Simulationssprachen bietet daher in diesem speziellen Fall keine Aufwandserleichterung. Die Basissimulationssprache müsste in diesem Fall Sprachelemente für partielle Differentialgleichungen bereitstellen.

Zelluläre Automaten in der Umweltmodellierung

In der Modellierung von Umweltphänomen verwendet man Simulationsmodelle, die Raum und Zeit berücksichtigen. Dabei haben sich zelluläre Automaten als ein geeignetes Konzept zur Modellierung etabliert. Ein zellulärer Automat ist Teil eines Systems, das aus vielen zellulären Automaten zusammengesetzt ist. Er hat eine lokale Sichtweise und ändert seinen Zustand in Abhängigkeit der Zustände seiner Nachbarn, die ebenfalls zelluläre Automaten sind. Ein zellulärer Automat besteht aus einer Beschreibung von Zustandsgrößen sowie einer Transitionsfunktion, die Zustandsänderungen beschreibt.

Theisselmann [75] entwickelt die DSL ECAL (Environmental Cellular Automata Language) für zelluläre Automaten, die speziell für die Umweltmodellierung geeignet sind. Er betrachtet zunächst existierende Spezialwerkzeuge und Modellierungsumgebungen in diesem Bereich. Um ein bestimmtes Problem zu lösen, müssen diese Werkzeuge häufig integriert werden. Die bereitgestellten Sprachen sind dabei ähnlich, unterscheiden sich jedoch in Details. Theisselmann schlägt den modellbasierten Ansatz EMS mit 3 Ebenen vor, der die Integration verschiedener Werkzeuge vereinfachen und adäquate Beschreibungen mit DSLs erlauben soll. Auf Ebene 2 befinden sich dabei ECAL-Modelle. ECAL bietet auf der einen Seite spezielle Sprachelemente für zelluläre Automaten. Auf der anderen Seite sind die Sprachelemente jedoch allgemein, da sich

zelluläre Automaten in verschiedensten konkreten Anwendungsbereichen beschreiben lassen. DSLs für diese konkreten Domänen befinden sich auf Ebene 1. Ein Beispiel ist eine DSL zur Beschreibung der Ausbreitung vom Bränden. Eine Ebene-1-DSL definierte eine adäquate Syntax. Die Semantik wird als Abbildung auf die Konzepte von ECAL definiert. Für ECAL existieren bereits Transformationen in die erforderlichen Spezialwerkzeuge und Simulationsumgebungen zur Analyse der Modelle.

ECAL ist ebenfalls eine textuelle DSL, da die Beschreibung der Zellen eines Automaten hauptsächlich aus Zustandsgrößen und Transitionsfunktionen aufgebaut ist. Zustandsgrößen lassen sich mit Variablen beschreiben. Transitionsfunktionen beschreiben Zustandsänderungen mit Anweisungen. Diese können an bestimmte Bedingungen geknüpft sein, die mit Ausdrücken beschrieben werden. Da Variablen, Anweisungen und Ausdrücke besonders gut textuell darstellbar sind, wurde ECAL offensichtlich als textuelle DSL konzipiert.

In ECAL sind weitere Sprachelemente enthalten, die häufig auch ein Teil von Simulationssprachen sind. Dazu gehören Operationen zur Listenverwaltung und zur Bestimmung zufälliger Werte entsprechend bestimmter Verteilungsfunktionen. Diese Sprachelemente wurden für ECAL ebenfalls definiert und konnten dabei, trotz ihrer allgemeinen Anwendbarkeit, nicht aus einer Mehrzwecksprache übernommen werden.

Modelle in ECAL sind ereignisorientiert beschrieben. Zustandsänderungen können zeitdiskret und zeitkontinuierlich erfolgen. Für die Simulation eines ECAL-Modells beschreibt Theisselmann eine Abbildung in das Java-Simulationsframework jDisco [33], das die Modellierung und Simulation zeitdiskreter, zeitkontinuierlicher als auch kombinierter Prozesse erlaubt. Die Verknüpfung mit räumlichen Daten erfolgt über das Framework Geotools[5], dass ebenfalls in Java implementiert ist. Die gemeinsame Implementierungssprache hat dabei den Integrationsaufwand erheblich reduziert.

Eine Beschreibung der drei Ebenen von EMS mit einer erweiterbaren Simulationssprache ist vorstellbar. DSLs auf Ebene 1 könnten als Erweiterungen von ECAL beschrieben werden. Dabei wird die Semantik als Abbildung nach ECAL definiert. ECAL auf Ebene 2 kann als Erweiterung der Basissimulationssprache beschrieben werden. Für die Anbindung an bestehende Frameworks wie z.B. Geotools muss ein entsprechender Mechanismus bereitstehen, der es erlaubt auf die Anwendungsschnittstelle einer anderen Software zuzugreifen. Diese Anforderung muss von einer erweiterungsbasierten Sprachentwicklungstechnik berücksichtigt werden. Oft existiert bereits Software, die nicht neu geschrieben kann, sondern eingebunden werden sollte.

Experimentieren

Wenn ein Simulationsmodell unter bestimmten Anfangswerten nachweisbar ein ähnliches Verhalten wie das Original aufweist, kann man dazu übergehen die Anfangswerte zu variieren um so zu neuen Erkenntnissen mit Hilfe des Modells zu gelangen. Diesen Vorgang bezeichnet man als Experimentieren. Der Vorgang kann mit einer Beschreibung seines Ablaufs exakt spezifiziert werden. Man kann dazu, wie bei vielen anderen Problemen auch, natürlich die allgemeinen Sprachelemente einer Programmiersprache einsetzen. Im Laufe der Zeit entstanden jedoch spezielle Sprachen zur Beschreibung

[5] http://www.geotools.org

von Abläufen, da man die immer wieder gleichen Prinzipien in der Beschreibung erkannte. Verbreitet sind hier grafische Workflow-Sprachen, deren Konzepte denen von UML-Aktivitäten ähneln.

Kühnlenz [41] konzipiert eine textuelle Sprache mit dem Namen ExpL zur Beschreibung von Experimentier-Workflows. Ein textuelle Syntax wurde gewählt, da ExpL-Beschreibungen deklarativ sind und deklarative Sprachen ebenfalls häufig textuelle Sprachen sind und weil sich textuelle Sprachen inklusive der notwendigen Werkzeuge mit weniger Aufwand entwickeln lassen als grafische Sprachen. Die Sprache ExpL wird in Fallstudien sowohl zum Experimentieren mit Nano-Strukturmodellen von Wider als auch mit ECAL-Modellen von Theisselmann in der Praxis eingesetzt.

Die Domäne von ExpL ist das Planen und Ausführen von Experimenten mit Modellen aus verschiedensten anderen Domänen. Typische Probleme, die dabei auftreten, werden durch die Sprachelemente von ExpL in verständlicher und prägnanter Form beschreibbar.

Bei Kühnlenz gibt es ein Problem. Experimentbeschreibung und Simulationsmodell werden mit separaten Techniken entwickelt. Das Experimentieren erfordert die Übergabe von Eingabedaten sowie die Entgegennahme und Auswertung von Ausgabedaten in Bezug auf das Simulationsmodell. Da das Simulationsmodell aber in einer beliebigen Sprache zur Simulation beschrieben sein kann, muss jedes Simulationsmodell zur Verarbeitung der Ein- und Ausgabedaten aus einer ExpL-Beschreibung speziell manuell vorbereitet werden. Dies erhöht den Aufwand beim Einsatz von ExpL.

ExpL definiert ein Typsystem, das dem von Programmiersprachen ähnelt. Es unterstützt die Typen Real, String, Integer und Boolean. Außerdem sind die komplexen Datentypen Point2d und Point3d zur Beschreibung von Punkten in einem Koordinatensystem enthalten. Das Typsystem ist dabei nur durch Änderungen am Sprachmetamodell erweiterbar. Komplexe Datentypen können nicht in einer ExpL-Beschreibung definiert werden, da die Sprache nicht das Konzept einer Klasse enthält. In einer objektorientierten Simulationssprache hätte man dieses Konzept direkt zur Verfügung.

Kühnlenz wünscht sich eine Erweiterung von ExpL, die allgemeine statistische Auswertungsmethoden enthält. Diese sind in Sprachen zur Simulation zeitdiskreter Systeme immer vorhanden. Eine Erweiterung einer Simulationssprache scheint daher vorteilhaft.

Auffällig in ExpL sind Sprachelemente, die direkt mit konkreten Werkzeugen und Technologien zusammenhängen, z.B. SVN und GIT zum Zugriff auf Simulationsmodelle, die unter Kontrolle einer Software zur Versionsverwaltung stehen. Ändern sich diese Technologien oder kommen neue hinzu, sind zwangsläufig Ergänzungen oder Änderungen an der Sprache notwendig.

Kühnlenz erwähnt ein mögliches Sprachelement TimelineDiagram zur Erzeugung eines Diagramms aus den Ein- und Ausgabedaten, das jedoch nicht Teil von ExpL ist. Diese Beispiele lassen erkennen, dass der Sprachentwickler von ExpL nicht jede mögliche Anwendung vorausahnen kann. Es wird immer wieder Anwendungen geben, die von neuen speziellen Sprachelementen profitieren können. Im Falle von ExpL kommt dabei erschwerend hinzu, dass es sich nicht um eine Mehrzwecksprache handelt, bei der man ein Sprachelement auch durch eine Menge von Funktionen oder Klassen beschreiben könnte. In ExpL ist man immer gezwungen, die Sprachdefinition anzupassen. Dies beinhaltet Änderungen am Metamodell, der Grammatik

sowie der Abbildung, die der Ausführung von ExpL-Beschreibungen dient. Jede dieser Änderungen erfordert ein spezielles Werkzeug mit Abhängigkeiten zu anderen Werkzeugen. Möchte ein Modellierer als Sprachentwickler diese Änderungen vornehmen, benötigt er Wissen und Erfahrung in all diesen Bereichen. Der Aufwand ist dabei nicht unerheblich.

Die Sprache ExpL hat Ähnlichkeiten mit ECAL. Beide Sprachen sind als Kernsprachen für eine Menge von Beschreibungen in einer bestimmten Domäne angelegt. Steigt die Notwendigkeit nach spezifischen Sprachelementen in einer spezielleren Domäne, so schlagen beide Ansätze die Einführung von DSLs vor, die über die Konzepte der Kernsprache hinausgehen. Während Theisselmann explizit von drei Ebenen spricht, sind diese bei Kühnlenz nicht klar erkennbar. Bei Kühnlenz wird eine DSL in die Sprachdefinition von ExpL eingefügt. Dies erfolgt bei der Anwendung von ExpL in der Domäne von ECAL. Die notwendigen Sprachelemente von ECAL werden zur Sprachdefinition von ExpL hinzugefügt. Bei Theisselmann wird eine DSL dagegen als eigenständige Sprache, unabhängig von ECAL definiert und es gibt eine Abbildung als Modelltransformation nach ECAL. Damit sind bei Theisselmann die Verantwortlichkeiten klarer getrennt.

Da sich ExpL um die Konzepte von ECAL erweitern lässt, ist anzunehmen, dass es eine Kernsprache gibt, die sowohl für ExpL als auch für eine ECAL eine geeignete Basis darstellt. Diese Kernsprache kann eine ereignis- oder eine prozessorientierte Simulationssprache sein. Die Übergabe von Daten zwischen Experiment und Simulationsmodell würde kein Problem darstellen, da beide in der selben Sprache beschrieben sind.

2.4 Sprachentwicklung

Der Einsatz einer Simulations-DSL setzt, wie der Einsatz jeder anderen Sprache auch, das Vorhandensein einer maschinenverarbeitbaren Beschreibung der Sprache sowie von Werkzeugen, in Form von Computerprogrammen, voraus, die für die Arbeit mit Modellen eingesetzt werden können. Damit die Entwicklung einer Sprache also effizient ist, muss die Sprache effizient beschreibbar sein und es müssen Werkzeuge effizient bereitgestellt werden können.

Ein Sprachentwicklungsansatz, der besonders effizient ist, aber auch besonders flexibel für die Definition verschiedenster Sprachen eingesetzt werden kann, ist der metamodellbasierte Ansatz. Dabei wird jeder Aspekt einer Sprache mit einer Spezialsprache in Bezug auf ein Metamodell beschrieben, das die grundlegende Struktur der Sprache definiert. Sprachwerkzeuge können für die einzelnen Aspekte automatisch abgeleitet werden. Bevor ich auf die Merkmale des metamodellbasierten Ansatzes eingehe, werde ich zunächst auf die Notwendigkeit von Sprachwerkzeugen eingehen.

2.4.1 Sprachwerkzeuge

Das einzige, aus theoretischer Sicht, tatsächlich erforderliche Sprachwerkzeug ist ein Compiler, der es erlaubt Programme in einer Sprache zu übersetzen und auszuführen. Die Erstellung von Programmen kann bei textuellen Sprachen mit einem beliebigen Text-Editor erfolgen. Die Fehleranalyse und die Auswertung einer Simulation können

mit einfachen Programmausgaben erfolgen. Diese rudimentären Mittel sind jedoch bei größeren Modellen nicht mehr ausreichend.

Heutige Entwicklungsumgebungen stellen umfangreiche Werkzeuge für die Programmierung bereit. Beispiele für solche Umgebungen sind Eclipse[6] und IntelliJ[7] für die Sprache Java. Wenn die Simulationsmodellierung in einer Programmiersprache erfolgt, so kann der Simulationsentwickler direkt von diesen Werkzeugen profitieren. Wird jedoch eine spezielle Simulationssprache verwendet, so werden Sprachwerkzeuge von einer ähnlichen Qualität erwartet, da sie die Modellierung und Analyse erheblich vereinfachen.

Ein textueller Editor sollte die Hervorhebung der syntaktischen Einheiten eines Programms (Syntaxhervorhebung) während der Eingabe sowie die kontextabhängige Bereitstellung von Vorschlägen für mögliche Vervollständigungen an einer bestimmten Stelle im Programm (Autovervollständigung) unterstützen. Die Bereitstellung dieser Funktionen ist immer abhängig von einer konkreten Sprache. Der Editor selbst oder das Programm, in das er integriert ist, sollte eine Ausführung des notierten Programms erlauben. Für das Debugging eines notierten Programms sollte es mit dem Editor möglich sein, Haltepunkte zu setzen, die aktuelle Programmposition während der Ausführung hervorzuheben, die Ausführung schrittweise fortzusetzen, sowie den Programmzustand untersuchen zu können. Das Debugging sollte in Bezug auf das notierte Programm möglich sein.

Die Erfüllung dieser elementaren Anforderungen ist von praktischer Bedeutung und erleichtert die Modellierung in einer Sprache. Jede neue Sprache sollte also Sprachwerkzeuge mit den entsprechenden Funktionen bereitstellen.

Am Beispiel von SLX kann man das Problem der Bereitstellung dieser elementaren Sprachwerkzeuge gut nachvollziehen. Die Sprache wird von einem sehr kleinen Team entwickelt. Dadurch sind die Sprachwerkzeuge nicht auf dem gleichen qualitativ hohen Niveau, das von heutigen Entwicklungsumgebungen vorgegeben wird. Der textuelle Editor von SLX unterstützt keine Syntaxhervorhebung während der Eingabe, sondern erst nachdem ein Programm übersetzt wurde. Es ist außerdem keine Autovervollständigung vorhanden. Obwohl SLX eine erweiterbare Sprache ist, gibt es kein spezielles Debugging für Programme, die Erweiterungen verwenden. Das Debugging ist immer nur auf dem Niveau der Basissprache SLX möglich. Im Gegensatz zu verbreiteten Entwicklungsumgebungen für Programmiersprachen, die als Open-Source bereitstehen und jedem eine Erweiterung der Umgebung erlauben, ist die Entwicklungsumgebung von SLX nicht als Open-Source verfügbar.

2.4.2 Sprachen

Computersprachen sind bei der Arbeit mit Computern allgegenwärtig. Jedes Programm, dass auf einem Computer läuft, ist in einer Computersprache geschrieben. Computersprachen sind Mittel zur Programmierung von Maschinen, so dass diese Berechnungen realisieren, die der Lösung von Problemen dienen. Insbesondere ist eine Computersprache aber auch eine Notation. So sehen es auch Aho et al. [1]:

[6] https://www.eclipse.org
[7] https://www.jetbrains.com/idea

> Programming languages are notations for describing computations to people and to maschines.

Programmiersprachen sind Computersprachen, die für die Beschreibung von Berechnungen verwendet werden. Nicht jede Computersprache dient jedoch der Berechnung. Dies trifft insbesondere auf DSLs zu, die lediglich der Strukturbeschreibung dienen. Scheidgen [62] definiert eine Computersprache daher allgemeiner:

> Languages are means to convey information to something. Computer languages are all those languages that are used to convey information to a computer.

Für Scheidgen besitzen Sprachen immer auch eine Syntax, die dem Notationsbegriff von Aho et al. entspricht.

> A language instance is built from language constructs instances: a language instance comprises (connected) construct instances that form a structure. This structure is the syntax of the language instance.

Eine Sprache legt im Allgemeinen Regeln fest nach denen Beschreibungen in der Sprache formuliert werden können und die deren Bedeutung (Semantik) festlegen. Beim Einsatz auf einem Computersystem spielen die unterschiedlichen Aspekte einer Sprache jeweils eine bestimmte Rolle. Zur besseren Auseinandersetzung mit Sprachen und ihren Aspekten ist es daher sinnvoll entsprechend eindeutige Begriffe einzuführen.

Scheidgen, der sich in seiner Arbeit mit der objektorientierten Beschreibung von Computersprachen beschäftigt, hat dazu einen Begriffskatalog eingeführt, an dem sich die vorliegende Arbeit orientiert.

Eine *Sprache* definiert eine Menge von *Sprachinstanzen*. *Sprachinstanzen* sind dabei die Vorkommen in einer *Sprache*. *Sprachinstanzen* besitzen eine klar definierte *Syntax* und *Semantik*. Zu einer *Sprache* gehört eine *Sprachbeschreibung*, die ein endliches System von Regeln definiert, mit denen entschieden werden kann, welches die Instanzen einer Sprache sind und welche Bedeutung diese haben.

Die Bausteine aus denen *Sprachinstanzen* zusammengesetzt sind werden als *Sprachelemente* bezeichnet. *Sprachelemente* werden durch *Elementdefinitionen* beschrieben. Ein *Sprachelement* ist also eine Instanz einer *Elementdefinition*. Eine *Elementdefinition* beschreibt eine Menge von *Sprachelementen*. Eine *Elementdefinition* kann Beziehungen zu anderen *Elementdefinitionen* besitzen. Die jeweiligen *Sprachelemente* können dann nur entsprechend dieser Beziehungen miteinander verbunden werden.

Die *Syntax* einer Sprache ist eine Menge von Regeln, die die Struktur von *Sprachinstanzen* festlegen. Der Begriff Syntax wird vorwiegend für Sprachen verwendet, deren Sprachinstanzen eine Struktur aus Wörtern und Sätzen bilden. Dazu gehören formale Sprachen in der Automatentheorie [66]. Sprachinstanzen einer formalen Sprachen bestehen hier aus Symbolen, die mit alphanumerischen Zeichen dargestellt werden. Die Syntax einer formalen Sprache wird durch eine Grammatik definiert, die festlegt, welche Wörter es in einer Sprache gibt und wie diese zu Strukturen in Form von Sätzen zusammengefügt werden können. Sätze können dabei wieder aus Sätzen bestehen. Eine Sprachinstanz ist ein Satz, dessen Syntax durch eine Grammatik definiert ist. Diese 1-zu-1 Beziehung ist jedoch in der praktischen Anwendung nicht mehr gegeben.

Spracheninstanzen müssen bei ihrem praktischen Einsatz in Sprachinstanzen einer Sprache übersetzt werden, die eine Maschine versteht. Ein solcher Übersetzungs- bzw. Compilerprozess kann aus vielen Schritten bestehen, der wiederum in Sprachen formuliert ist. Zur besseren Verarbeitung werden die zu übersetzenden Sprachinstanzen dabei in einer geeigneten Form dargestellt. Ein zweckmäßige Form ist ein Baum, der die Satzstruktur und die Wörter in hierarchischer Form enthält und zugreifbar macht. Einen solchen Baum bezeichnet man als *abstrakten Syntaxbaum* (AST).

Wird eine objektorientierte Programmiersprache für die Übersetzung verwendet, so besteht ein AST in einem Programm aus einer Objektstruktur. Diese abstrahiert von Details der Darstellung, die für die Verarbeitung nicht relevant sind. Der AST ist also eine programmiertechnisch verarbeitbare Darstellung, die aus einer menschenlesbaren Darstellung entsteht. Ein AST kann ebenfalls mit einer Grammatik beschrieben oder aus dieser abgeleitet sein. Er kann jedoch auch auf eine ganz andere Weise beschrieben sein. Zur Unterscheidung bezeichnet man daher die Definition von Regeln für eine menschenlesbare Darstellung als *konkrete Syntax* und die für eine programmiertechnisch verarbeitbare Darstellung als *abstrakte Syntax*.

2.4.3 Ein Metamodell als Basis

Um das Jahr 2000 entstand ein neuer metamodellbasierter Entwicklungsansatz für Sprachen. Das neue an diesem Ansatz war die Beschreibung der abstrakten Syntax einer Sprache mit objektorientierten Konzepten. Eine solche objektorientierte Beschreibung wird hier als Metamodell bezeichnet. Ein Metamodell wird für die Beschreibung einer Sprachen und ihrer Vorkommen verwendet und ist dabei selbst auch ein Vorkommen einer anderen Sprachen, einer Metamodellierungssprache, für die ebenfalls eine Syntax und eine Semantik definiert werden muss.

Das Metamodell ist im Kern eine abstrakte objektorientierte Struktur und besitzt zunächst keine konkrete Darstellung. Für die Arbeit mit Metamodellen ist jedoch eine konkrete Darstellung zwingend erforderlich. Deshalb gibt es, je nach Einsatzzweck, verschiedene konkrete Darstellungsformen. Dazu gehören unter anderem

- eine grafische Darstellung, die die Notation der Unified Modelling Language (UML) verwendet, und als menschenlesbare Darstellung eingesetzt wird,
- eine Darstellung in XML für die Persistierung auf Maschinen, und
- eine Darstellung in einer objektorientierten Programmiersprache für die Entwicklung von Sprachwerkzeugen.

Die Metamodellierungssprache, die ja selbst auch eine Sprache ist, wird bei diesem Ansatz mit ihren eigenen Konzepten beschrieben. Da die Sprache aber nur die abstrakte Syntax beschreiben kann, wird die Semantik von einem Framework festgelegt. Ein Metamodellierungs-Framework definiert die Mechanismen, die für die Arbeit mit Metamodellen und Instanzen von Metamodellen notwendig sind. Ein Metamodellierungs-Framework kann abstrakt oder konkret sein.

Ein weit verbreitetes abstraktes Metamodellierungs-Framework ist die Meta Object Facilty (MOF). MOF ist in UML und in natürlicher Sprache spezifiziert. Ein konkre-

tes Metamodellierungs-Framework erlaubt es nun, Metamodelle als Objektstrukturen entsprechend ihrer Semantik zu erstellen und zu verwalten.

Ein verbreitetes konkretes Metamodellierungs-Framework ist das Eclipse Modelling Framework (EMF) für Java, das auf dem von der Object Management Group (OMG) standardisierten aber abstrakten Framework Essential Meta Object Facility (EMOF), einer Teilmenge von MOF, basiert und Ecore als Metamodellierungssprache verwendet.

Die metamodellbasierte Definition löst dabei auch das Problem der effizienten Verfügbarkeit von Sprachwerkzeugen [23]. Diese können in einer objektorientierten Programmiersprache unter Verwendung einer objektorientierten abstrakten Syntaxbeschreibung implementiert werden und profitieren dabei von der objektorientierten Konzeptbasis, die beiden Beschreibungen zugrundeliegt.

Beschreibung der abstrakten Syntax mit einem Metamodell

Ein Metamodell ist eine abstrakte Syntax, die Regeln mit objektorientierten Ausdrucksmitteln definiert. Eine Metamodell-Instanz ist eine Objektstruktur, die sich in objektorientierten Sprachen gut weiterverarbeiten lässt. Der Einsatz von Metamodellen ist heute weit verbreitet. Dies hat mehrere Gründe. Wird die abstrakte Syntax mit einem objektorientierten Metamodell beschrieben, so lassen sich Metamodell-Instanzen gut in einer objektorientierten Sprache weiterverarbeiten. Auf diese Weise werden z.B. Interpreter oder Transformatoren implementiert. Ein weiterer positiver Aspekt von Metamodellen sind die objektorientierten Ausdrucksmittel, die es erlauben Teile von Metamodellen wiederverwendbar zu beschreiben. Ein Beispiel bildet die UML Infrastructure [48], die bis UML Version 2.4.1 ein Teil der Spezifikation war. Die UML Infrastructure wurde sowohl beim Metamodell für UML (der sogenannte UML Superstructure) als auch beim Metamodell für MOF [50] eingesetzt. Im Zuge von Bestrebungen die UML-Spezifikation zu vereinfachen, wurde die Trennung zwischen Infrastructure und Superstructure mit UML Version 2.5 aufgehoben.

Wird ein Metamodell für die abstrakte Syntax einer Sprache eingesetzt, so lassen sich Strukturen beschreiben, die bei der Definition mehrerer ähnlicher Metamodelle eingesetzt werden können. Metamodelle haben sich außerdem als das zentrale Sprachartefakt durchgesetzt, auf deren Grundlage weitere Sprachaspekte beschrieben werden. Weitere Sprachaspekte sind die konkrete Syntax und die Semantik. Diese Sprachaspekte stehen dabei in Beziehung zu einem Metamodell.

Modellbegriff im Kontext von Metamodellen

In der Gemeinschaft, die sich mit Metamodellen beschäftigt, hat sich der Begriff *Modell* als Bezeichnung für eine Metamodell-Instanz durchgesetzt. Auf den ersten Blick, scheint der Begriff durchaus geeignet. In der Computersimulation ist ein Modell ein eingeschränktes Abbild eines relevanten Ausschnitts eines Systems, das die Grundlage für Experimente darstellt. Diese Definition ist passend, wenn die Sprache, zu der das Metamodell gehört, eine Sprache zur Systemmodellierung ist. Dies trifft jedoch nicht auf alle Sprachen zu. Ein Beispiel ist die Sprache MOF [50], die zur Beschreibung der Struktur von Metamodellen eingesetzt wird. Eine Sprachinstanz von MOF ist also ein Metamodell, das im Wesentlichen aus Klassen, Attributen und Assoziationen besteht.

Es ist jedoch schwer vorstellbar, dass ein Metamodell als Grundlage für Experimente dient. Die Frage ist: Was ist das System, das ein Metamodell nachbildet und wozu dienen Experimente mit einem Metamodell?

Ein Metamodell ist eine abstrakte Syntax, die die Struktur von Sprachinstanzen festlegt. Die Funktion des Metamodells ist demnach die Beschreibung der erlaubten Strukturen. Es abstrahiert dabei von Notationen in einer konkreten Syntax. Zu den Systemelementen gehören Klassen und Assoziationen. Diese erbringen durch ihre Wechselwirkung den Systemzweck. Entfernt man eine der Klasse, kann die Funktion nicht mehr erbracht werden, da nun eine kleinere Sprachinstanzmenge beschrieben wird.

Ein Bezug zum Modellbegriff kann also, selbst bei einer Strukturbeschreibung wie einem Metamodell gefunden werden. Dennoch wird in der Gemeinschaft der Simulation die Auffassung vertreten, dass der Begriff Modell als Bezeichnung für die Instanzen von Metamodellen im Allgemeinen nicht geeignet ist. Ein Modell ist hier eine Nachbildung eines Systems aus der uns umgebenden Umwelt, sei es existierend oder erdacht. Eine Metamodell-Instanz kann mit dieser Umwelt nicht in Beziehung gesetzt werden. Aus diesem Grund, trenne ich in dieser Arbeit die Begriffe strikt voneinander. Ein Modell bezeichnet immer eine Abstraktion im Sinne der Computersimulation, die sowohl aus der Beschreibung von Strukturen als auch von Verhalten besteht.

Beschreibung weiterer Sprachaspekte

Alle weiteren Aspekte einer Sprache werden unter Bezugnahme auf die Elemente des Metamodells in anderen Spezialsprachen, sogenannnten Sprachbeschreibungssprachen, ausgedrückt. Für jeden Aspekt einer Sprache gibt es dabei eine Spezialsprache. Die Spezialsprachen sind selbst auch metamodellbasiert beschrieben. Dadurch ist es möglich Sprachwerkzeuge unabhängig von einer konkreten Sprache auf der Basis des Metamodells für eine Spezialsprache zu implementieren. Das Sprachwerkzeug operiert auf einer konkreten Instanz dieses Metamodells, das einen Teilaspekte einer spezifizierten Sprache beschreibt, und für diesen Teilaspekt eine konkrete Funktionalität bereitstellt.

Weitere Aspekte einer Sprache sind

1. die konkrete Syntax,
2. die statische Semantik,
3. die dynamische Semantik und
4. das Debugging.

Das Debugging betrachte ich als eine besondere Form der dynamischen Semantik, die es erlaubt die Programmausführung zu unterbrechen und dabei Programmzustände zu untersuchen.

Die prinzipielle Bereitstellung von Sprachwerkzeugen durch Codegenerierung oder Interpretation ist für jeden dieser Teilaspekte, die ich den nachfolgenden Abschnitten betrachte, bereits untersucht.

Beschreibung der konkreten Syntax

Aus einer kontextfreien Grammatik, die mit einem Metamodell verknüpft ist, kann ein Parser inklusive eines textuellen Editors mit Syntaxhervorhebung und Autovervollständigung durch Interpretation der Grammatik erzeugt werden. Eine entsprechende Umsetzung wird von Scheidgen [62] beschrieben und ist in Form des Textual Editing Framework (TEF) realisiert.

Die Definition einer kontextfreien Grammatik erfolgt in TEF mit der Sprache *Textual Syntax Language* (TSL). Diese erlaubt die Formulierung einer EBNF[8]-ähnlichen attributierten kontextfreien Grammatik. Die Attribute der Grammatik enthalten Verweise auf Metaklassen und Metaattribute, über die Beziehungen zwischen Textfragmenten und Metamodellinstanzen ausgedrückt werden, um so während des Parsens, ein Modell entsprechend eines angegebenen Metamodells zu instanziieren und mit Objekten und primitiven Werten zu befüllen.

Ein andere konkrete Realisierung erfolgt im Rahmen des Frameworks Xtext[9]. Mit Xtext ist es möglich, aus einer kontextfreien Grammatik ein Java-Programm für einen Parser sowie für einen textuellen Editor zu generieren. Diese Werkzeuge können jedoch im Gegensatz zu TEF erst verwendet werden, nachdem der erzeugte Java-Code übersetzt ist. Anders als TEF erlaubt Xtext zusätzlich die Ableitung eines Metamodells aus einer kontextfreien Grammatik.

Beschreibung der statischen Semantik

Statische Semantik kann mit der Object Constraint Language (OCL) definiert werden. OCL erlaubt die Beschreibung von Invarianten für Metamodellinstanzen auf der Basis eines Metamodells. Ein Constraint-Checker für OCL (z.B. Eclipse OCL[10] oder Dresden OCL[11]) ist dann in der Lage, eine Metamodellinstanz auf die Erfüllung der hinterlegten Invarianten zu prüfen.

Beschreibung der dynamischen Semantik

Die dynamische Semantik beschreibt die Ausführung einer Sprachinstanz und wird deshalb auch als Ausführungssemantik bezeichnet. Sie lässt sich auf viele verschiedene Art und Weisen festlegen. Eine Klassifikation von Möglichkeiten zur Semantikbeschreibung wäre sehr umfangreich und liegt nicht im Fokus dieser Arbeit. Ich gebe deshalb nur eine eine grobe Unterteilung und konzentriere mich auf die Semantikbeschreibungen, die auch bei Programmiersprachen zum Einsatz kommen. Bei diesen Sprachen liegt ein Schwerpunkt auch immer auf einer laufzeiteffizienten Ausführung.

Die Ausführungssemantik einer Sprache lässt sich grob auf zwei Arten beschreiben: durch Interpretation oder durch eine Übersetzung in eine andere ausführbare Sprache. Bei der Interpretation wird eine Maschine beschrieben, die Sprachinstanzen verarbeiten kann. Je nachdem, wie diese Maschine eine Sprachinstanz verarbeitet,

[8] Erweiterte Backus-Naur-Form
[9] http://www.eclipse.org/Xtext
[10] https://projects.eclipse.org/projects/modeling.mdt.ocl
[11] https://github.com/dresden-ocl

wird die interpretative Semantik weiter unterteilt, und zwar in denotationale Semantik und operationale Semantik. Eine operationale Semantik definiert eine schrittweise Ausführung. Sie besteht aus einer Menge von Konfigurationen und einem System von Transitionen, das den Übergang von einer Konfiguration zur nächsten Konfiguration beschreibt. Ein Konfiguration besteht dabei jeweils aus der aktuellen Programmposition und Werten von Variablen.

In der metamodellbasierten Sprachentwicklung wird die Menge der Konfigurationen einer operationalen Semantik mit einem Metamodell festgelegt. Das Transitionssystem kann in einer anderen ausführbaren Sprachen beschrieben werden. Hier gibt es viele verschiedene Möglichkeiten. Scheidgen und Soden setzen UML-Aktivitäten [63] [71] ein, während Sadilek einen Ansatz beschreibt [58] bei dem beliebige existierende Programmiersprachen und sogar ausführbare Abstract State Machines (ASMs) [16] eingesetzt werden können. Diese Arbeiten wenden die operationale Semantik nicht explizit für Simulationssprachen an. Scheidgen und Soden erwähnen jedoch Simulationssprachen und weisen auf die prinzipielle Möglichkeit der Beschreibung quasiparalleler Prozesse und eines Prozessterminkalenders hin. In [56] geht man noch einen Schritt weiter und definiert die Ausführungssemantik einer einfachen Simulationssprache mit einer operationalen Semantik. Die Arbeiten problematisieren die laufzeiteffiziente Ausführung nicht, obwohl diese von großer praktischer Bedeutung ist.

Eine Sprache deren Semantik operational beschrieben ist, kann die Laufzeiteffizienz einer übersetzten Sprachen mit einer transformationalen Semantik nicht erreichen. Stellen wir uns ein einfaches Programm in einer imperativen Sprache vor, das aus einer Folge von Anweisungen besteht. Eine operationale Semantik beschreibt hier unter anderem Programmpositionen und den Fortschritt der Programmausführung durch das Verändern der Programmposition. Für die Ausführung der operationalen Semantik wird wiederum eine Maschine benötigt, bei der es sich letztendlich um ein Programm in einer anderen ausführbaren Sprache handeln muss. Dieses Programm, das die operationale Semantik unserer Beispielsprache ausführt, wird selbst auch auf einer konkreten Maschine ausgeführt. Dabei handelt es sich heute ausnahmslos um Rechner nach der Von-Neumann-Architektur. Der Rechner hat selbst einen Speicher, in dem sich Programme und Daten befinden. Er verwaltet die Ausführungsposition eines Programms und interpretiert seine Befehle. Eine operationale Semantik kann nur indirekt mit einem anderen Programm, das als Interpreter fungiert, ausgeführt werden. Eine transformationale Semantik kann dagegen eine Abbildung auf eine andere Sprache mit einer Übersetzung auf Befehle eines Rechners vornehmen. Diese direkte Ausführung durch den Rechner ist laufzeiteffizienter als die indirekte Ausführung eines Programms, dass die Befehle einer Sprachinstanz interpretiert.

Für die Beschreibung von Simulationssprachen ist nur eine Transformationssemantik zweckmäßig. Dies zeigt auch das Beispiel der Sprache SLX, deren Programme eine Laufzeiteffizienz aufweisen, die von keiner anderen Sprache erreicht wird. Ein Programm in SLX wird direkt in ein Assemblerprogramm abgebildet. Kapitel 4 führt Laufzeituntersuchungen von SLX und anderen Sprachen bei deren Einsatz für Simulationen durch und zeigt, dass SLX einen Prozesskontextwechsel fünfmal schneller als die laufzeiteffizienteste bekannte Simulationsbibliothek in C++ durchführen kann. Die vorliegende Arbeit setzt daher für die Definition der Semantik einer Simulationssprache eine transformationale Semantik ein.

Beschreibung des Debuggings

Für das Auffinden von Fehlern in Programmen wird ein Debugger benötigt. Dieser erlaubt es, das Programm anzuhalten, den Laufzeitzustand zu untersuchen und das Programm schrittweise auszuführen oder fortzusetzen. Die Darstellung des Laufzeitzustands sowie die Möglichkeiten zur Programmsteuerung sind von der jeweiligen Sprache des untersuchten Programms abhängig. Da der Debugger Einfluss auf die Ausführung eines Programms nimmt, ist die Beschreibung der Ausführungssemantik einer Sprache maßgeblich für die Implementierung des Debuggers. Ein Interpreter wird erweitert, so dass der intern für die Ausführung verwaltete Laufzeitzustand zugreifbar ist und die Ausführung der nächsten Aktionen an bestimmten Stellen angehalten werden kann. Bei einer Übersetzung müssen Instruktionen im Zielprogramm hinzugefügt werden, so dass die Ausführung an bestimmten Stellen unterbrochen und der Laufzeitzustand ausgelesen werden kann. Die Beschreibung eines Debuggers für eine metamodellbasierte Sprache mit einer operationalen Semantik wird in [15] untersucht.

2.4.4 Erweiterung einer bestehenden Sprache

Heutige Simulationssprachen sind nur unzureichend anpassbar an domänenspezifische Anwendungsszenarien. Zwar können Klassen und Objekte zu objektorientierten Sprachen hinzugefügt werden, jedoch fehlt eine angepasste Werkzeugunterstützung und eine prägnante und kompakte Notation für die neuen Konzepte. Deshalb muss für jede neue Domäne zunächst ein Sprachentwickler die domänenspezifische Sprache und die notwendigen Werkzeuge implementieren, bevor der Anwender sie einsetzen kann.

SLX als einzige erweiterbare Simulationssprache

Der einzige bekannte Vertreter einer Simulationssprachen, die um neue Sprachkonzepte mit einer eigenen Notation erweitert werden kann, ist die Sprache Simulation Language with Extensibility (SLX), die von Wolverine Software als proprietäre Software, bestehend aus der Sprache und einer Entwicklungsumgebung vertrieben wird.

Der Erweiterungsansatz erlaubt nur die Beschreibung von regulären Sprachen und ist damit nicht geeignet komplexe Strukturen mit Mehrfachangabe von Sprachelementen zu definieren. In [10] werden die Sprachbeschreibungsmöglichkeiten von SLX untersucht. Der Laufzeitkern von SLX ist hochlaufzeiteffizient, aber nicht offen zugänglich und somit auch nicht erweiterbar. SLX bietet einen Texteditor, der jedoch keine unmittelbare Unterstützungen für Erweiterungen bietet.

2.5 Laufzeitaspekte von Simulationssystemen

Eine Sprache zur Simulation definiert lediglich die Syntax und Semantik von Sprachelementen. Die Sprache allein ist dabei nicht ausreichend um eine Computersimulation durchzuführen. Dazu wird ein *Simulationssystem* benötigt, dass aus folgenden Komponenten besteht:

1. einer Sprache zur Simulation,
2. einem Modelleditor zur Eingabe und Bearbeitung von Modellen,
3. häufig gebrauchten Funktionen in Form einer Bibliothek z.B. zur Erzeugung von Zufallszahlen,
4. einer Implementierung der Next-Event-Methode zur Abarbeitung von Ereignissen und zum Voranschreiten der Modellzeit, und
5. einem Compiler oder Interpreter, der Sprachinstanzen entsprechend der implementierten Next-Event-Methode ausführt.

Zusätzlich ist es sinnvoll, einen Debugger bereitzustellen, der es erlaubt die Abarbeitung eines Modells schrittweise durchzuführen und dabei den Zustand zu betrachten um mögliche Fehler im Modell zu erkennen.

Die Laufzeit einer Simulation hängt von vielen Faktoren ab und kann für eine Untersuchung entscheidend sein. Denn es macht durchaus einen Unterschied, ob eine Simulation nach einigen Stunden oder erst nach Tagen ein Ergebnis liefert. Die Komplexität eines Modells und der notwendige simulierte Realzeitraum sind Einflussfaktoren, die sich aus der konkreten Untersuchung ergeben und vom Simulationssystem nicht beeinflusst werden können. Der spezifische Einflussfaktor eines Simulationssystems auf die Laufzeit ist die Art und Weise wie die einzelnen Komponenten zur Abarbeitung der Simulation implementiert sind. Dabei spielt sowohl die algorithmische Umsetzung von Berechnungen als auch die gewählte Implementierungssprache eine bedeutende Rolle.

Ein Algorithmus mit einem hohen Einfluss auf die Laufzeit ist der gewählte Scheduling-Algorithmus. Dieser dient der Bestimmung des nächsten Zeitereignisses und der Einordnung der zukünftigen Zeitereignisse. Dazu wird eine Datenstruktur benötigt, deren Effizienz an diesen beiden Operationen gemessen wird. Es gibt zahlreiche Untersuchungen verschiedener Scheduling-Algorithmen [76] [68]. Zusammenfassend gibt es keinen besten Scheduling-Algorithmen, der in jeder Situation allen anderen überlegen ist. Die Effizienz hängt immer vom konkreten Modell ab, d.h. welche konkreten Zeitereignisse durch den Algorithmus verarbeitet werden müssen. Man kann jedoch sagen, dass es Scheduling-Algorithmen gibt, die in den meisten Modellen schneller sind als andere.

Ein weiterer Einflussfaktor neben Zeitereignissen sind Zustandsereignisse. Ein Zustandsereignis tritt ein, sobald eine Zustandsgröße einen bestimmten Wert besitzt. Diese Situation lässt sich mit einer Bedingung formulieren, die Bezug auf die Zustandsgröße nimmt. Ändert sich nun der Wert einer Zustandsgröße, so muss die Bedingung geprüft werden. Im positiven Fall tritt das Zustandsereignis ein.

Die Laufzeit für die Auswertung von Zustandsbedingungen hängt von ihrer Implementierung ab. Die Sprache UML definiert Zustandsereignisse als semantischen Variationspunkt und lässt ihre konkrete Implementierung offen. Eine Variante der Implementierung ist es die Zustandsbedingung nach jeder ausgeführten Aktion erneut zu prüfen. Diese Variante ist jedoch nicht besonders laufzeiteffizient. Eine andere Möglichkeit ist es die Zustandsbedingung nur dann zu prüfen wenn eine explizit anzugebende Prüfaktion ausgeführt wird. Eine solche Aktion müsste als Teil einer Aktionsfolge immer nach der Änderung einer relevanten Zustandsgröße eingefügt werden.

Diese Variante ist laufzeiteffizienter, da die Überprüfung gezielter erfolgt. Der Nachteil ist jedoch eine Anfälligkeit für das Fehlen der Prüfaktion. Dieses Problem kann mit einer Variante automatisiert werden, in der eine entsprechende Prüfaktion nach jeder Zuweisung an eine beteiligte Zustandsgröße automatisch erfolgt. Diese Art der Implementierung wird von SLX mit der wait-until-Anweisung umgesetzt.

Einen hohen Einfluss hat außerdem der Teil eines Simulationssystems, der zur Ausführung von Modellen eingesetzt wird. Dabei kann ein Interpreter zum Einsatz kommen, der selbst ein Programm ist, das in einer Implementierungssprachen geschrieben ist und das Modell schrittweise abarbeitet. Durch diese Indirekte Ausführung des Modells durch die Ausführung des Interpreters, der das Modell zunächst analysiert, sind Interpreter häufig langsamer als Compiler in der Ausführung. Eine Compiler übersetzt das Modell in eine andere Sprache, für die es wiederum einen Interpreter oder einen Compiler geben muss. Am Ende einer solchen Kette steht eine Maschinensprache, die durch die Prozessorarchitektur des verwendeten Computers vorgegeben ist.

In einem prozessorientierten Modell gibt es einen weiteren Einflussfaktor. Während der Ausführung der Prozessbeschreibung wird immer wieder auf das Eintreten von Ereignissen gewartet, die von anderen Prozessen oder durch den Vorschritt der Zeit ausgelöst werden. Da in der sequentiellen Simulation, immer nur ein Prozess gleichzeitig aktiv sein kann, muss die Abarbeitung der Prozessbeschreibung unterbrechbar sein. Bei einer Unterbrechung müssen alle Daten, die die Ausführung des aktuellen Prozesses betreffen gesichert werden und die Daten inklusive der Ausführungsposition des nächsten Prozesses müssen wiederhergestellt werden. Diesen Vorgang bezeichnet man als Kontextwechsel zwischen Prozessen.

Kontextwechsel können einen erheblichen Einfluss auf die Laufzeit einer Simulation haben. In einem prozessorientierten Modell gibt es im allgemeinen viele Prozesse mit Wechselwirkungen untereinander. Durch diese Wechselwirkungen kommt es zu eher häufigeren Kontextwechseln. Die Ausführungszeit eines Kontextwechsels ist abhängig von der eingesetzten Sprache zur Simulation und von der Implementierung der Kontextwechsel im Simulator. Der Anteil an der Gesamtlaufzeit ist abhängig vom Einfluss anderer Laufzeitaspekte.

In einer Sprache wie GPSS, besteht ein Modell lediglich aus einer Sequenz von Blöcken, die von Transaktionen durchlaufen werden. Transaktionen können dabei direkt zu einem bestimmten Block springen. Es gibt jedoch nicht das Konzept einer Funktion. Außerdem besitzen Transaktionen keine flexible Attributstruktur. Bei einem Kontextwechsel in einem GPSS-Model ist also lediglich die aktuelle Position der Transaktionen zu sichern bzw. wiederherzustellen.

In Programmiersprachen ist ein Kontextwechsel wesentlich umfangreicher. Prozessbeschreibungen erfolgen hier in Koroutinen. Dabei kann eine Koroutine weitere Koroutinen aufrufen. Die Aufruffolge der Koroutinen inklusive aller Funktionsparameter und der lokalen Variablen sind dabei für den Kontextwechsel zu berücksichtigen. Des Weiteren wird die Prozessstruktur in objektorientierten Sprachen durch Klassen beschrieben. Das Objekt des aktuellen Prozesses mit seinen Attributwerten ist deshalb ebenfalls für den Kontextwechsel von Bedeutung. Da Programmiersprachen im Allgemeinen nicht auf Kontextwechsel quasi-paralleler Prozesse optimiert sind, benötigt ein Kontextwechsel hier mehr Zeit als ein Kontextwechsel in einer Assemblerähnlichen Sprache wie z.B. GPSS.

3 Eine laufzeiteffiziente Simulationsbasissprache

3.1 Einleitung

Die Grundlage jedes Ansatzes zur Spracherweiterung stellt eine Basissprache dar. Sie legt die Konzepte fest, die in jede Erweiterung integrierbar sind und damit die Definition einer neuen Sprache erleichtern. Diese Basiskonzepte sind außerdem der Ausgangspunkt für die Erweiterbarkeit. Sie definieren die Erweiterungspunkte, an denen die Sprache um zusätzliche syntaktische Ausprägungen ergänzt werden kann. Für die neuen syntaktischen Formen wird eine Semantik festgelegt, so dass im Verbund mit der Basissprache ein semantisch gültiges Programm entsteht.

Die Basissprache kann entweder viele Konzepte enthalten oder sie kann auf einen kleinen Konzeptkern beschränkt sein. Dabei gibt es eine Grundregel: Jedes für die prozessorientierte Modellierung notwendige Konzept, das nicht in der Basis enthalten ist, muss sich durch eine Erweiterung beschreiben lassen. Die Basis kann jedoch Konzepte enthalten, die nicht notwendigerweise in ihr enthalten sein müssten. Die Notwendigkeit eines Konzeptes ergibt sich aus den Anforderungen, die durch die Anwendung der Sprache vorgegeben sind. Es gibt also einen gewissen Spielraum im Umfang der Basissprache, der von bestimmten Anforderungen abhängt.

Dieses Kapitel beschäftigt sich mit der Frage, welche Konzepte eine objektorientierte Basissprache für die prozessorientierte Simulation enthalten sollte, so dass Simulations-DSLs sowohl effizient entwickelt werden können als auch besonders laufzeiteffizient ausführbar sind. Aus diesen beiden allgemeinen Anforderungen leite ich eine Reihe von speziellen Anforderungen an eine Basissprache ab. Eine Basissprache, die diese Anforderungen erfüllt, enthält die passenden Konzepte und entspricht somit der gesuchten Sprache.

Folgende Anforderungen muss die Basissprache erfüllen:

1. die in Kapitel 2 der Arbeit aufgeführten Anforderungen an Simulationssprachen sind erfüllt (Simulationskonzepte),
2. eine prozessorientierte Modellierung ist möglich,
3. DSLs mit Beziehungen zwischen den Sprachelementen sind mit Erweiterungen beschreibbar, die außerdem bestehende Basiselemente in ihrer Beschreibung wiederverwenden können (prägnante Modelle),
4. Simulationen, die mit den beschriebenen DSLs durchgeführt werden, sind besonders laufzeiteffizient ausführbar (laufzeiteffiziente Ausführung),
5. bestehende Implementierungen in C++ und in Java können wiederverwendet werden.

Es gibt keinen bestehenden Ansatz mit einer Basissprache, die diese Anforderungen erfüllt. Es gibt jedoch ähnliche Sprachen, die Teilanforderungen umsetzen. Die von mir konzipierte Basissprache bezeichne ich als Discrete-Event Base Language (DBL). Um die Erfüllung der Anforderungen für DBL zu bewerten, vergleiche ich die Sprache mit vorhandenen ähnlichen Sprachen. Dazu gehören die Basissprache BL von MPS, Xcore von Xtext und SLX von Wolverine.

Da die Simulation immer nur eines von vielen Mitteln der Modellanalyse, aber nicht den einzigen Zweck einer Modellierung darstellt, muss außerdem diskutiert werden, in wie weit sich ein solcher Ansatz für eine Verwendung der Modelle in realen Anwendungen einsetzen lässt.

Das Kapitel beschäftigt sich zunächst mit der Frage, was eine erweiterbare Sprache eigentlich ist. Im Anschluss werden die im vorliegenden Kontext notwendigen Erweiterungskonzepte, die Grundvoraussetzung für die Erweiterbarkeit einer Sprache sind, beschrieben. Danach wird die schrittweise Hinzunahme weiterer Konzepte im Hinblick auf die genannten Anforderungen diskutiert. Dabei werden jeweils die Vor- und Nachteile betrachtet.

3.2 Was ist eine erweiterbare Sprache?

Standish [72] fasst 1975 die ersten Bestrebungen Sprachen erweiterbar zu machen, die sich bereits auf das Jahr 1960 zurückführen lassen, in einer einfachen und umfassenden Definition zusammen:

> Simply put, extensibility permits programming language users to define new language features.

Erweiterbarkeit erlaubt also dem Anwender einer Programmiersprache neue Sprachmerkmale zu definieren. Dazu steht dem Anwender eine Basissprache mit verschiedenen Erweiterungsmechanismen zur Verfügung. Mit den Erweiterungsmechanismen können neue Notationen, neue Datenstrukturen, neue Operationen oder auch neue Kontrollanweisungen zur Basissprache hinzugefügt werden. Nach Standish erlaubt eine erweiterbare Sprache außerdem die Anpassung, also die Änderung, von Sprachmerkmalen. Er sieht Anwender prinzipiell in der Lage durch Spracherweiterung, Konzepte einzuführen, mit denen sich algorithmische Problemlösungen in speziellen Anwendungsfeldern prägnant, klar und ohne unnötige Details beschreiben lassen.

Welche Sprachen gehören nach dieser Definition zu den erweiterbaren Sprachen und wie unterscheiden sie sich in ihren Erweiterungsmechanismen? Zingaro [86] gibt dazu 2007 einen Überblick über moderne erweiterbare Sprachen. Er konzentriert sich auf imperative, objektorientierte und funktionale Mehrzwecksprachen. Zingaro betrachtet Makros in C, eine erweiterbare Variante von Java, OCaml und Scheme, und vergleicht die Einschränkungen durch die verwendeten Parsing-Mechanismen sowie deren Ausdrucksmöglichkeiten.

Im Bereich Simulation wird 2000 von James O. Henriksen die erweiterbare Simulationssprache SLX [34] konzipiert. Im Gegensatz zu anderen erweiterbaren Sprachen, bietet SLX eine komplette Entwicklungsumgebung, die sowohl Sprachentwicklung als auch Simulationsdurchführung vereinfacht. Der Erweiterungsmechanismus in SLX lässt jedoch nur einfache Erweiterungen auf dem Niveau von regulären Sprachen zu.

In der metamodellbasierten Sprachentwicklung entsteht das Meta Programming System (MPS) [77]. Es handelt sich dabei um eine Entwicklungsumgebung für Sprachen, die es erlaubt verschiedene Aspekte einer Sprache mit speziellen Sichten zu beschreiben. MPS verarbeitet die einzelnen Aspektbeschreibungen und generiert eine sprachspezifische Entwicklungsumgebung. Den Ausgangspunkt für den Erweiterungsmechanismus in MPS stellt das Metamodell der Basissprache BL dar. Die Sprache entspricht weitestgehend der Sprache Java. Eine DSL definiert selbst ein Metamodell, dessen Klassen Spezialisierungen zu Klassen aus dem BL-Metamodell festlegen. MPS stellt auf der Basis einer konkreten Syntaxbeschreibung einen projektionalen Modelleditor bereit.

Neben MPS gibt es weitere aktuelle Arbeiten [22], die zeigen, dass das Thema erweiterbare Sprachen auch heute noch relevant ist. Diese Arbeiten betrachten jedoch nicht die speziellen Anforderungen, die prozessorientierte Simulationssprachen an eine Basissprache stellen.

3.3 Erweiterungskonzept in DBL

DBL stellt ein Erweiterungskonzept bereit, das eine syntaktische Ergänzung von Basiselementen erlaubt, die auf Elemente in der Sprache DBL zurückgeführt werden. Dabei sind Erweiterungen auf mehreren Stufen möglich. Eine Erweiterung kann aus anderen Erweiterungen zusammengesetzt sein. Jede Erweiterung muss jedoch vollständig auf Basiselemente zurückführbar sein.

Eine Erweiterung wird durch einen Verweis auf ein Basiselement definiert. Für dieses Basiselement legt die Erweiterungen eine weitere mögliche Ausprägung fest. Die Erweiterung kann an allen Stellen in einem Basisprogramm verwendet werden, an denen auch das Basiselement stehen kann.

Eine Erweiterung besteht aus der Definition einer konkreten Syntax und einer Abbildung auf die Basissprache. Die konkrete Syntax wird mit einer attributierten kontextfreien Grammatik festgelegt. Die Attribute dienen als Grundlage für die Ableitung einer abstrakten Syntax, so dass in der Beschreibung der Semantik einer Erweiterung, die konkrete Erweiterungsinstanz analysiert werden kann. Für jedes erweiterbare Basiselementes gibt es eine Basisgrammatikregel. Die Grammatik der Erweiterung definiert eine Startregel, die als eine alternative Ausprägung zur Basisgrammatikregel hinzugefügt wird. Damit muss eine Erweiterung immer syntaktisch unterscheidbar von der Basissprache und von anderen Erweiterungen sein.

Die Semantik wird als Ausführungssemantik in DBL selbst festlegt. Sie beschreibt eine Abbildung auf DBL mit Anweisungen in DBL. Zusätzlich gibt es spezielle Anweisungen, die eine Ersetzung der Erweiterung durch DBL-Elemente festlegen. Die Ersetzung kann an der Stelle erfolgen, an der die Erweiterung steht oder ein Element an einer beliebigen anderen Stelle im Modell einfügen. Durch die Ausführung der Abbildung entsteht ein DBL-Teilprogramm, das die Erweiterung ersetzt.

Dieses Erweiterungskonzept ist ausreichend mächtig, um die Konzepte einer prozessorientierten Simulationssprache zu beschreiben, sofern die Basissprache bestimmte notwendige Konzepte enthält. Der nachfolgende Abschnitt beschäftigt sich mit diesen notwendigen Konzepten. Danach wird die Hinzunahme weiterer Konzepte diskutiert, die prinzipiell durch Erweiterung möglich ist. Dabei dient SLX als Referenz, da

SLX die einzige bekannte erweiterbare Simulationssprache ist. Die allgemeine Mächtigkeit des Erweiterungskonzeptes, seine Syntax und Semantik, sowie seine Einschränkungen sind dagegen Teil von Kapitel 5.

3.4 Allgemeinsprachliche Basiskonzepte

Bekannt ist, dass sich Simulationen mit Programmiersprachen beschreiben lassen. Programmiersprachen sind universell einsetzbare Turing-vollständige Sprachen. Die Basissprache muss also zumindest Turing-vollständig sein.

Turing-Vollständigkeit erreicht man bereits mit wenigen Konzepten. Ein aus der Berechenbarkeitstheorie bekanntes Beispiel sind GOTO-Programme. Dabei handelt es sich um spezielle Programme, die lediglich aus fünf Anweisungen zusammengesetzt sind: i) Zuweisung an eine Variable, vermehrt um eine Konstante, ii) Zuweisung vermindert um eine Konstante, iii) Sprunganweisung, iv) bedingte Sprunganweisung und v) Stop-Anweisung. Eine solche Sprache als Basis wäre theoretisch bereits ausreichend, um auch Simulationen zu beschreiben.

Aus der Turing-Vollständigkeit einer Sprache ergibt sich jedoch keine Aussage über den Aufwand für die Implementierung eines Programms oder seine Laufzeit. Diese beiden Aspekte sind aus praktischer Sicht aber sehr relevant. Die Laufzeit ist abhängig von den Eigenschaften der Maschine auf der das Programm ausgeführt wird. Die heutigen Computersysteme, die nach dem Von-Neumann-Prinzip arbeiten, sind in einer passenden Maschinensprache programmierbar. Ein Maschinenprogramm besteht dabei aus Instruktionen, die Operationen durchführen und auf einen Speicher zugreifen. Maschinenprogramme sind jedoch für Menschen nicht direkt lesbar und so gibt es Assemblersprachen, die verständliche Abkürzungen für die Instruktionen der Maschine enthalten und deren Programme auf Maschinenprogramme abgebildet werden. Dennoch orientiert sich die Formulierung eines Assembler-Programms sehr nah an der Funktionsweise der Maschine und nur wenig am Problem.

Mit einer Assembler-Sprache lassen sich jedoch sehr laufzeiteffiziente Programmausführungen realisieren, da der Programmierer das Programm für eine besonders effiziente Ausführung auf der speziellen Maschine optimieren kann. Eine solche manuelle Optimierung ist jedoch nicht unproblematisch. Die Lösung des Problems ist im Programm nicht mehr leicht nachvollziehbar und Änderungen oder Erweiterungen am Programm sind damit schwierig. Außerdem ist nicht jeder Programmierer auch gut in der Optimierung von Programmen.

Deshalb gibt es heute Hochsprachen, die Konzepte für eine leichte Nachvollziehbarkeit und Wartbarkeit von Programmen bereitstellen, und gleichzeitig mit automatisch optimierenden Compilern ausgestattet sind, die Programme in besonders laufzeiteffizient ausführbare Maschinenprogramme übersetzen. Die heute verbreiteten Hochsprachen sind objektorientierte Sprachen. Besonders stark verbreitet sind die Sprachen Java und C++.

Die Sprache C++ ist dabei auf die Möglichkeit, Programme besonders effizient zu implementieren ausgelegt und enthält dafür ein Zeiger-Konzept, das es erlaubt Werte im Speicher abzulegen, zu referenzieren, zu entnehmen und den Speicher wieder freizugeben. Der Programmierer übernimmt die Verwaltung seiner Daten im Speicher selbst. Dies erlaubt zwar besonders effiziente Programme, hat jedoch den Nach-

teil, dass die Programmierung fehleranfällig ist. C++ wird besonders dann eingesetzt, wenn die Effizienz in Bezug auf die Laufzeit oder den Speicher besonders wichtig sind.

Die Sprache Java stellt dagegen abstraktere Konzepte als C++ bereit. So gibt es lediglich Objektreferenzen und die Speicherverwaltung wird durch das Java-Laufzeitsystem automatisch realisiert. In Java lassen sich Programme im Allgemeinen leichter implementieren. Es gibt eine Vielzahl von Programmen, insbesondere auch für die Sprachentwicklung, die in Java programmiert sind. Dazu gehören Metamodellierungs-Frameworks wie EMF und AMOF [62], sowie Entwicklungsumgebungen wie EProvide [58], MPS [38] [77], Spoofax [39], Xtext [36] und TEF [62]. Eine Anbindung an bereits bestehende Werkzeuge für die Sprachentwicklung ist für den in dieser Arbeit entwickelten Ansatz von Bedeutung, damit eine prototypische Implementierung des Ansatzes leicht möglich ist. Die Basissprache muss daher so konzipiert sein, dass eine Anbindung an bestehende Programme in Java möglich ist.

Als Ausgangspunkt jeder Modellierung ist eine Basissprache mit den bekannten Konzepten für eine strukturierte Programmierung aus Hochsprachen sinnvoll. Dazu gehören insbesondere Funktionen, Variablen, Anweisungen und Ausdrücke. Die Basissprache muss sich außerdem im Rahmen des hier entwickelten Ansatzes nach Java und nach C++ abbilden lassen. Die Abbildung nach Java dient dabei nur der Anbindung bestehender Modellierungs- und Analysetools. Dagegen wird die Abbildung nach C++ für die laufzeiteffiziente Ausführung von Simulationen eingesetzt. Beide Zielsprachen müssen bei der Konzeption berücksichtigt werden.

Sollte die Basissprache aber selbst bereits objektorientierte Konzepte enthalten oder sollten diese durch eine Erweiterung beschreibbar sein? Betrachtet man die Sprache C als eine Sprache ohne objektorientierte Konzepte und die Sprache C++, die C um diese entsprechenden Konzepte erweitert, so kann davon ausgegangen werden, dass eine entsprechende Definition durch Erweiterungen prinzipiell möglich ist. Der Compiler für C++ enthält jedoch unter anderem auch automatische Optimierungen für den Zugriff auf Objektstrukturen, von denen die Basissprache DBL, bei einer Definition von objektorientierten Konzepten als Erweiterung nicht profitieren könnte. Für eine Abbildung auf laufzeiteffizient ausführbare Programme, sollte die Basissprache deshalb objektorientierte Konzepte bereits enthalten.

3.5 Objektorientierte Basiskonzepte

Die Integration der Vorteile von C++ hinsichtlich laufzeiteffizient ausführbarer Programme und der Vorteile von Java in Bezug auf die Wiederverwendung bestehender Tools für die Sprachentwicklung lässt sich durch eine Basissprache erreichen, die objektorientierte Konzepte enthält, die sich sowohl auf C++ als auch auf Java abbilden lassen. Die Semantik der Basissprache kann so als Abbildung auf diese beiden bereits bestehenden Sprache, für die es jeweils eine Semantik gibt, definiert werden. Dabei muss jedoch sichergestellt werden, dass sowohl die Abbildung für C++ als auch die Abbildung für Java die gleiche Semantik festlegen. Da dieses Problem im Allgemeinen nicht gelöst werden kann, beschreibe ich die Semantik von DBL in den nachfolgenden Abschnitten informal und diskutiere die Konzepte, die sich in ihrer Abbildung nach C++ und nach Java unterscheiden.

Die abstrakte Syntax von DBL definiere ich mit einem Metamodell. Die konkrete Syntax ist mit einer kontextfreien Grammatik festgelegt. Ich beschreibe nun die Syntax und die Semantik der Konzepte der Basissprache und stelle diese ausschnittsweise mit dem Metamodell und der Grammatik dar. Das Konzept für die Erweiterung werde ich aufgrund des Umfangs hier noch nicht erklären und erst im Kapitel 5 vorstellen. Ich beschränke mich also auf Konzepte, die aus C++ und Java bekannt sind und diskutiere deren Aufnahme in die Basissprache.

Da die Anwendbarkeit des Ansatzes DMX an einer Sprache für Zustandsautomaten gezeigt werden soll, muss die Basissprache DBL in ihrer Syntax und Semantik vollständig definiert sein. Je mehr Konzepte die Sprache enthält, desto größer wird der notwendige Aufwand für ihre Definition. Deshalb beschränke ich die Sprache an verschiedenen Stelle und nehme Konzepte, die als Erweiterung nicht beschreibbar sind und zur Basissprache dazugehören sollten, nicht in die Basissprache auf.

Es gibt eine ganze Reihe von Konzepten in DBL, die enthalten sind, weil die automatischen Optimierungen der Compiler von C++ und Java wiederverwendet werden sollen. Dazu gehören Variablen, Funktionen, Klassen, Interfaces und Objekte, sowie Anweisungen und Ausdrücke.

3.5.1 Module

Ein Programm in der Sprache DBL kann auf mehrere Dateien aufgeteilt sein. Jede Datei besteht aus einem Modul, das Module aus anderen Dateien einbinden kann. Ein Modul definiert einen Namensraum für bestimmte Konzepte in einer Datei, ähnlich wie Packages in Java und Namespaces in C++. Ein Modul kann globale Variablen und Funktionen, Klassen, Interfaces und Erweiterungen enthalten (siehe Abb. 3.1).

Ein Modul kann nicht verschachtelt werden. Beim Einbinden eines anderen Moduls, sind alle Bezeichner aus diesem Modul im aktuellen Modul unter Angabe ihres Namens sichtbar. Nur falls das aktuelle Modul und das eingebundene Modul ein Konzept mit dem gleichen Namen enthalten, muss Eindeutigkeit über die Angabe des eingebundenen Moduls hergestellt werden.

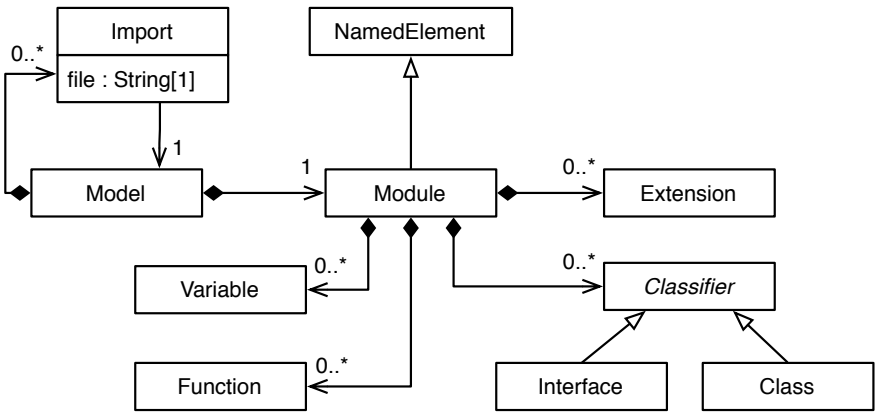

Abbildung 3.1: Moduldefinition im DBL-Metamodell.

3.5.2 Funktionen

Funktionen gibt es in mehreren Ausprägungen: als globale Funktionen, als Methoden von Klassen, als Methoden von Objekten und als Konstruktoren für Objekte. Außerdem gibt es eine ausgewiesene globale Funktion mit dem Namen main, die als Programmeintrittspunkt dient. Abb. 3.2 zeigt das DBL-Metamodell für Funktionen.

Globale Funktionen müssen nicht notwendigerweise zu DBL dazugehören. Die Sprache Java zeigt, dass globale Funktionen auch als statische Methoden von Klassen möglich sind. Die Definition einer globalen Funktion kann als eine Erweiterung für Modulinhalte erfolgen. Diese Erweiterung wird dann auf eine statische Methode in einer Klasse abgebildet.

Beim Aufruf dieser globalen Funktion gibt es jedoch ein Problem. Auch der Ausdruck für den Aufruf einer Funktion müsste erweitert werden, so dass der Aufruf einer globalen Funktion auf den Aufruf der entsprechenden statischen Klassenmethode abgebildet wird. Da eine Erweiterung immer syntaktisch unterscheidbar von der Basissprache und von anderen Erweiterungen sein muss, kann der Aufruf für globale Funktionen nur durch ein spezielles Schlüsselwort realisiert werden.

Ein spezielles Schlüsselwort für den Aufruf einer globalen Funktion würde man jedoch nicht erwarten, da dies in anderen Programmiersprachen auch nicht notwendig ist. Eine andere Möglichkeit, bei der die gleiche Syntax wiederverwendet werden kann, ist die Einführung eines Konzeptes zur Operatorüberladung, ähnlich wie in C++. Damit kann eine globale Funktion auf den Funktions-Operator in einer neuen Klassen abgebildet werden. Die Funktion wird zu einem Objekt, das mit den Funktions-Operator aufgerufen wird.

Für ein Konzept zur Operatorüberladung als Teil von DBL müsste jedoch auch eine Abbildung nach Java festgelegt werden. Um den Aufwand an dieser Stelle zu reduzieren, wird auf eine Hinzunahme eines Konzeptes zur Operatorüberladung verzichtet. Die Operatorüberladung ist außerdem hinreichend genau untersucht und in anderen Sprachen bereits vorhanden. Somit gehören globale Funktionen zu DBL dazu.

Statische Methoden müssten nicht notwendigerweise in DBL enthalten sein. Eine statische Methode kann auch immer auf die Erzeugung eines Objektes mit anschließendem Aufruf einer Objektmethode abgebildet werden. Ein solches Vorgehen wäre jedoch nicht laufzeiteffizient. Außerdem wäre eine Anbindung von statischen Methoden in C++ und Java nicht möglich. Eine solche Anbindung ist für die Wiederverwendung bestehender Funktionen jedoch notwendig. Statische Methoden gehören daher zu DBL dazu.

3.5.3 Variablen

Variablen treten als globale Variablen, als Attribute von Klassen, als Attribute von Objekten, als Parameter von Funktionen und als lokale Variablen in Funktionen auf (siehe Abb. 3.2). Variablen existieren in diesen verschiedenen Ausprägungen in DBL, da diese auch in Java und C++ vorhanden sind. Listing 3.1 zeigt ein Beispielprogramm, das die verschiedenen Arten von Funktionen und Variablen enthält.

```
#import "stdlib"

module m;

// globale Variable
Counter c;

// globale Funktion main
void main() {
    // Konstruktoraufruf
    c = new Counter();

    // Methodenaufruf
    SystemOut.println("Counter with ID=" + c.getId());
}

class Counter {
    // Objektattribut
    int id;

    // statisches Klassenattribut
    static int count;

    // Konstruktor
    new() {
        id = count;
        count = count + 1;
    }

    // Objektmethode
    void getId() {
        return id;
    }

    // statische Klassenmethode mit Parameter value
    static void increaseCount(int value) {
        // lokale Variable result
        int result = count + value;
        return result;
    }
}
```

Listing 3.1: DBL-Beispielprogramm für Funktionen und Variablen

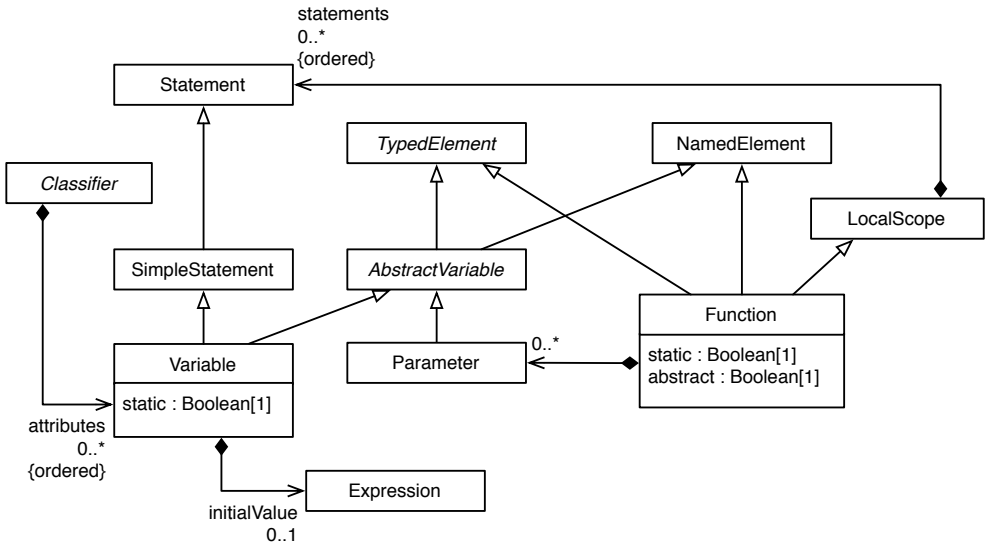

Abbildung 3.2: Definition von Funktionen und Variablen im DBL-Metamodell.

3.5.4 Typen

Die Typen von DBL unterteilen sich in primitive Typen und in Classifier mit den Ausprägungen Klasse und Interface (siehe Abb. 3.3). Die primitiven Typen sind: int für ganze Zahlen, double für Fließkommazahlen, boolean für Wahrheitswerte, string für Zeichenketten und void als Typ von Funktionen ohne Rückgabewert. Kollektortypen, die verschiedene Arten von Mengen abbilden, wie z.B. Listen oder Sets, müssen mit Hilfe von Klassen und Interfaces beschrieben werden. Auf Felder von Typen wird in DBL zur Vereinfachung verzichtet, obwohl diese in C++ und Java vorhanden sind. Ein DBL-Element, das einen Typ hat, enthält entweder einen primitiven Typ oder einen speziellen Ausdruck, der auf einen Classifier verweist. Listing 3.2 zeigt die Verwendung der verschiedenen Typen in einem DBL-Beispielprogramm.

```
1  module types;
2
3  int i = 2;
4  double d = 2.5;
5  boolean b = true;
6  string s = "text";
7
8  List l = new ArrayList;
9
10 class A {}
11
12 void main() {
13     A a = new A();
14   l.add(new A());
15 }
```

Listing 3.2: DBL-Beispielprogramm für Typen.

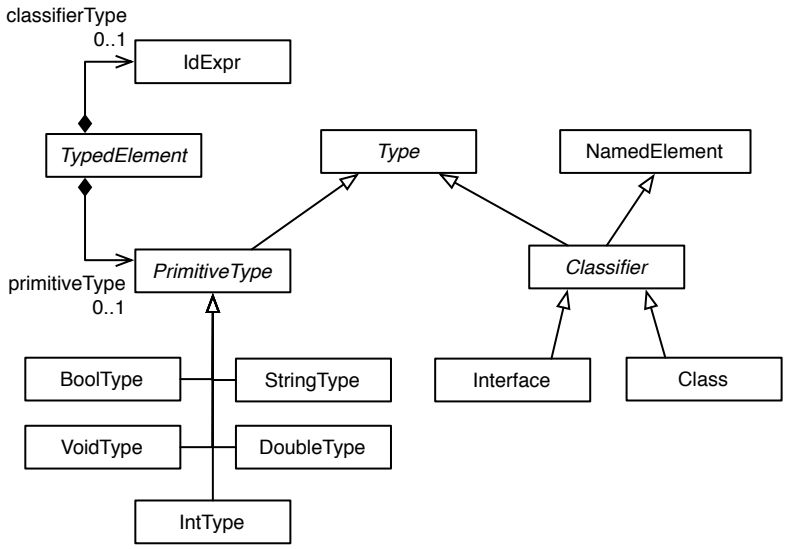

Abbildung 3.3: Definition von Typen im DBL-Metamodell.

3.5.5 Klassen, Interfaces und Objekte

Klassen können Konstruktoren, Attribute und Methoden enthalten (siehe Abb. 3.4). Einen Destruktor gibt es nicht. Eine Klasse kann von höchstens einer anderen Klassen erben. Sie kann jedoch viele Interfaces implementieren. Ein Interface kann selbst von vielen anderen Interfaces erben. Dabei kann ein Interface lediglich abstrakte Methoden und statische Attribute enthalten. Diese Art der Vererbung entspricht der Vererbung aus Java. Sie lässt sich einerseits besonders einfach nach Java abbilden und andererseits auch nach C++ abbilden. Listing 3.3 zeigt die Verwendung von Klassen und Interfaces.

Im Gegensatz zu Java können Interfaces Konstruktoren deklarieren. Eine Klasse, die ein Interface implementiert, muss auch die angegebenen Konstruktoren implementieren. Dieses neue Konzept ist für die Anbindung bestehender Klassen aus Java und C++ notwendig.

Die Werte von Objektvariablen können polymorph zum definierten Typ sein. Methodenaufrufe werden dabei dynamisch auf Basis des tatsächlichen Objekttyps aufgelöst. Attributzugriffe werden statisch auf Basis des definierten Typs der Variable aufgelöst.

Die Weitergabe von Objekten erfolgt mit einer Referenzsemantik, die auch von Java verwendet wird. Der Speicher, den Objekte dabei belegen, wird von einem Laufzeitsystem automatisch verwaltet. Ein DBL-Programm muss also keinen Speicher alloziieren und freigeben. Für primitive Typen erfolgt die Weitergabe von Werten mit Wertesemantik.

Eine Referenzsemantik wird gewählt, da diese die Modellierung vereinfacht und diese Vereinfachung die Vorteile einer laufzeiteffizienteren Implementierung überwiegen. Die Sprache SLX, die ebenfalls eine Referenzsemantik verwendet, zeigt, dass der Einfluss dieser auf die Laufzeit nicht entscheidend ist.

```
module m;

interface I1 {
    static int i; // abstrakte Methode
    void m();
    new(string s); // Konstruktordeklaration
}
interface I2 {
    void m();
}
interface I3 extends I1,I2 {
    void n();
}

class B {
    string s;
    new(string s) {
        self.s = s;
    }
}

// Klasse A erbt von Klasse B und implementiert die Interfaces I1 und I3
class A extends B implements I1,I3 {
    new() {
        self("A"); // Weiterleitung an anderen Konstruktor
    }
    new(string s) {
        super(s); // Weiterleitung an Basisklassenkonstruktor
    }
}

void main() {
    I1 i1 = new A();
    i1.m();
}
```

Listing 3.3: DBL-Beispielprogramm für Klassen und Interfaces.

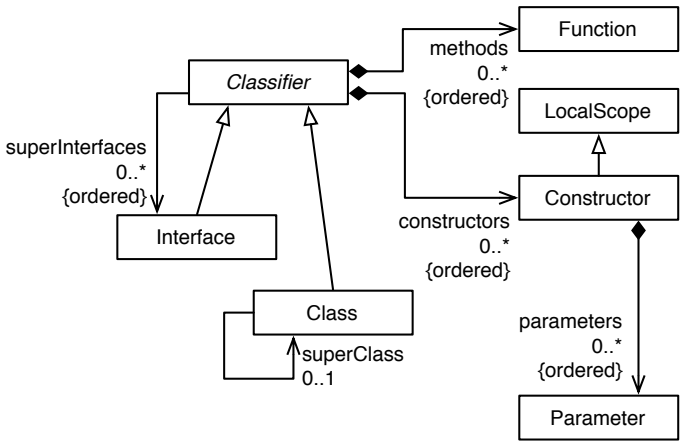

Abbildung 3.4: Definition von Classifiern im DBL-Metamodell.

3.5.6 Anweisungen

DBL enthält Anweisungen zur Definition von lokalen Variablen, für die Zuweisung von Werten zu Variablen, für den Aufruf von Funktionen und für die Rückgabe des Wertes einer Funktion. Als strukturierte Anweisungen sind außerdem enthalten: die If-Anweisung, die Switch-Anweisung mit den Anweisungen Break und Continue, die While-Schleife und die zählerbasierte For-Schleife. Der Teil des Metamodells, der Anweisungen definiert, ist in den Abb. 3.5, 3.6 und 3.7 dargestellt. Ein Beispielprogramm, das die Notation zeigt, ist in Listing 3.4 zu sehen.

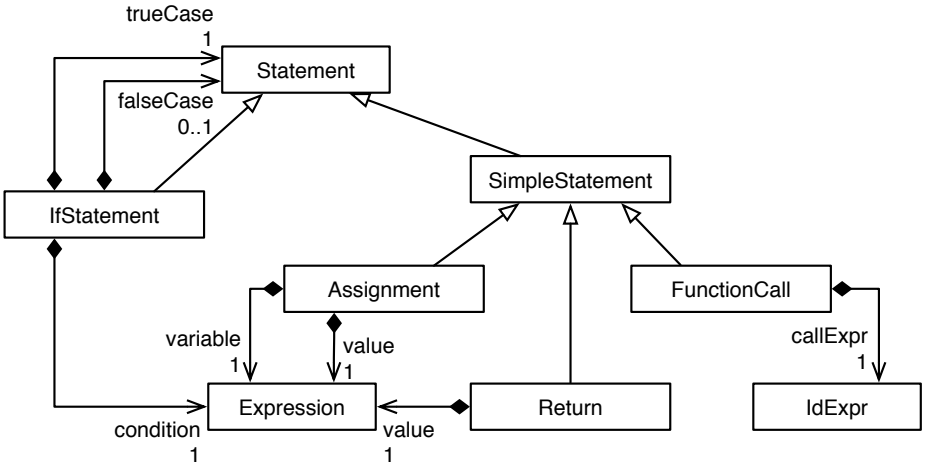

Abbildung 3.5: Definition von Anweisungen im DBL-Metamodell (Teil 1).

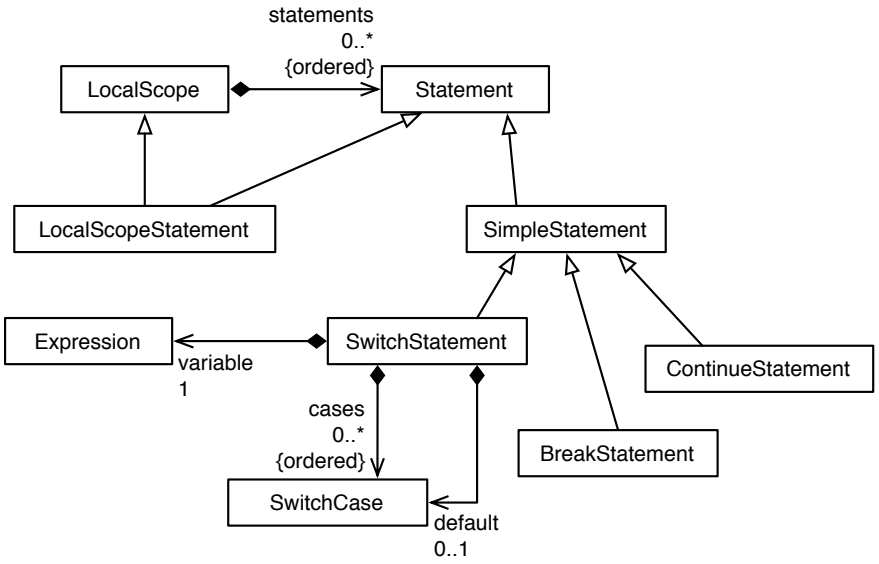

Abbildung 3.6: Definition von Anweisungen im DBL-Metamodell (Teil 2).

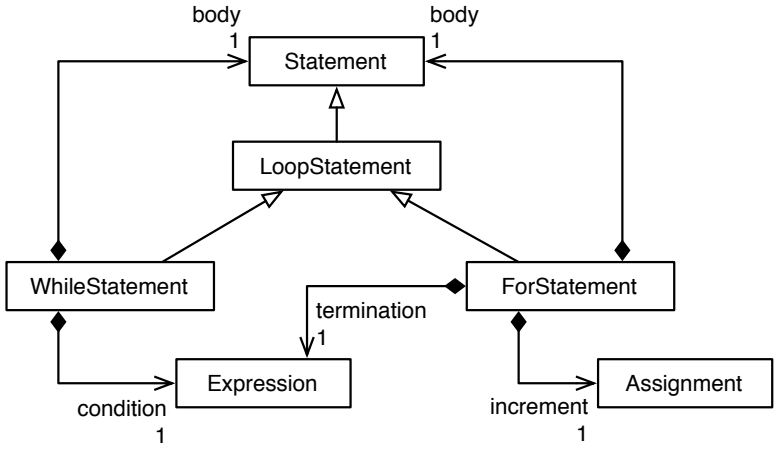

Abbildung 3.7: Definition von Anweisungen im DBL-Metamodell (Teil 3).

```
int f() {
  f(); // Funktionsaufruf

  int i; // lokale Variable
  i = 1; // Zuweisung

    // Switch-Anweisung
  switch (i) {
    case 1: print "1";
    default: { print "default"; }
  }

    // bedingte Schleife
  while (i > 0) {
    continue;
  }

    // zählerbasierte Schleife
  for (i = 0; i < 1; i = i+1) {
    break;
  }

    // Bedingung
  if (i > 1) {}
  else if (i < 1) {}
  else {}

    // Rückgabe
  return 1;
}
```

Listing 3.4: DBL-Beispielprogramm für Anweisungen.

3.5.7 Ausdrücke

Die Sprache enthält eine Reihe von Ausdrücken, die mit Operatoren zu komplexen Ausdrücken kombiniert werden können. Dazu gehören arithmetische, relationale, boolesche und Objekt-bezogene Operatoren (create, instanceof, cast), Variablenzugriffe, Funktionsaufrufe, eine Reihe von Literalen und eine Reihe von vordefinierten Bezeichnern (self, super). Ein Ausschnitt des DBL-Metamodells, der Ausdrücke definiert, wird in Abb. 3.8 gezeigt. Das Metamodell definiert für jede Prioritätsklasse i eine eigene Metaklasse $L(i)Expr$. So kann eine Erweiterung für einen Ausdruck einer bestimmten Prioritätsklasse definiert werden. Die mögliche Navigation entlang von Bezeichnern (Instanzen von NamedElement) wird mit der Metaklasse IdExpr festgelegt. Eine Navigation ist dabei ein Baum von IdExpr-Instanzen. Die Notation von Ausdrücken entspricht weitgehend der Notation von Ausdrücken in Java.

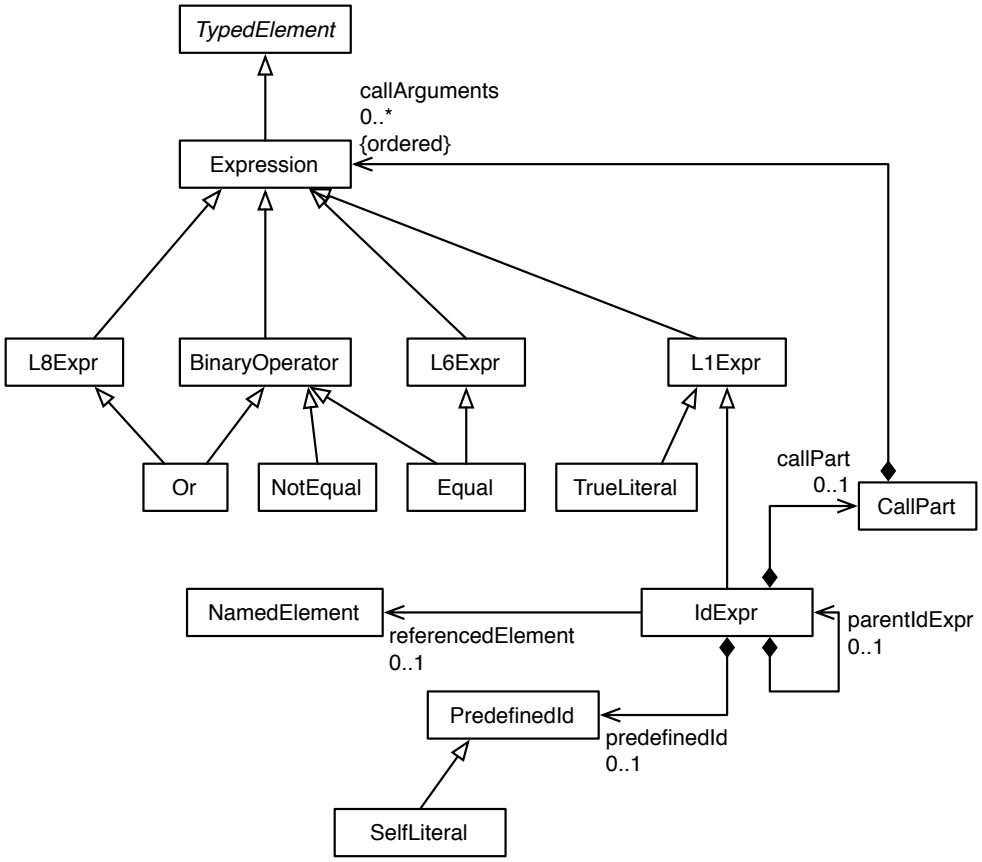

Abbildung 3.8: Ausschnitt der Definition von Ausdrücken im DBL-Metamodell.

3.5.8 Nicht enthaltene Konzepte

Andere Sprachkonzepte, die es in Java und C++ gibt, sind nicht enthalten. Dazu gehören die Ausnahmebehandlung, die Möglichkeit zur Verwaltung von Klassen in Namensräumen, die Vergabe von Sichtbarkeiten, Typparameter, der Aufzählungstyp enum, Konstanten und Methodenüberladung. Diese Konzepte sind nicht enthalten, um den Umfang der vorliegenden Arbeit beherrschbar zu halten.

3.6 Konzepte zur prozessorientierten Modellierung

3.6.1 Einleitung

In einem prozessorientierten Modell werden Ereignisse und Aktionen, die ein bestimmtes Systemelement betreffen, zusammenhängend beschrieben. Eine prozessorientierte Beschreibung besteht aus einer Abfolge von Anweisungen, die auf den Eintritt eines Ereignisses warten, ein Ereignis auslösen oder den Modellzustand verändern. Die Realisierung einer solchen Beschreibung ist ein Prozess.

Im Allgemeinen gibt es mehrere Prozesse, die parallel existieren und die parallele Abläufe im realen System nachbilden. Um die Reproduzierbarkeit von Simulationsläufen sicherzustellen, muss eine deterministische Reihenfolge in der Ausführung der parallelen Prozesse im Modell sichergestellt werden. Dies wird in sequentiellen Simulationssystemen durch eine quasi-parallele Ausführung in Bezug auf eine Modellzeit gewährleistet.

Prozesse, die zu einem Modellzeitpunkt im realen System parallel ablaufen, werden nacheinander nach bestimmten deterministischen Regeln ausgeführt. Danach wird die Modellzeit erhöht und der Vorgang wiederholt sich bis alle Prozesse vollständig abgearbeitet sind.

3.6.2 Prozessmodellierung in bestehenden Sprachen

Ein Konzept für eine prozessorientierte Modellierung mit quasi-paralleler Ausführung ist weder in C++ noch in Java als Sprachkonzept enthalten. In beiden Sprachen lassen sich Prozesse jedoch mit einer Bibliothek modellieren. Beim Bibliotheksansatz ist man jedoch in der Implementierung einer laufzeiteffizienten Ausführung beschränkt. Im Gegensatz dazu enthält die Simulationssprache SLX ein Konzept für quasi-parallele Prozesse als Teil der Sprache selbst. Damit ist es möglich, eine Abbildung mit einer hochlaufzeiteffizienten Ausführung zu beschreiben. Im Fall von SLX, erfolgt die Abbildung in eine Assemblersprache.

Bei der Ausführung quasi-paralleler Prozesse ist die Zeit für den Wechsel zwischen den Prozessen von großer Bedeutung. Da sich bei einem Wechsel auch der Kontext, bestehend aus den sichtbaren Variablen und den aufgerufenen Funktionen ändern kann, wird der Prozesswechsel auch als Kontextwechsel bezeichnet.

3.6.3 Umfang einer Teilsprache zur Prozessmodellierung

In SLX gibt es 14 Scheduling-Anweisungen[1] mit denen das Verhalten von Prozessen beschrieben werden kann. Ein Prozess ist in SLX ein aktives Objekt, das durch eine aktive Klasse beschrieben wird. Objekte, die kein Verhalten definieren, werden als passive Objekte bezeichnet. Diese Objekte sind durch normale Klassen definiert.

Im Folgenden beschäftige ich mich mit der Frage, welche der Scheduling-Anweisungen von SLX notwendig sind, um eine laufzeiteffiziente Ausführung zu erreichen und welche der Anweisungen in der Sprache selbst beschrieben werden können, um so den Umfang der Basissprache zu reduzieren. Ich beginne mit grundlegenden Anweisungen zum Prozesswechsel und diskutiere die Hinzunahme weiterer Anweisungen, die Zeit- und Zustandsereignisse betreffen.

Prozesswechsel

Der Wechsel vom einem Prozess zu einem anderen Prozess lässt sich bereits mit einer bedingten Sprunganweisung realisieren, die sowohl in Assemblersprachen als auch in der Sprache C vorhanden ist. Sobald ein Prozess auf ein Ereignis wartet, wird der nächste auszuführende Prozess bestimmt, die aktuelle Ausführungsposition gespeichert und zur gespeicherten Position des gewählten Prozesses gewechselt.

Wird eine Prozessbeschreibung jedoch mit Hilfe von Funktionen strukturiert, ist zusätzlich der Funktionsaufruf-Stack zu sichern und wiederherzustellen. In diesem Fall ist eine Sprunganweisung allein nicht mehr ausreichend. Stattdessen kommen spezielle Funktionen zum Einsatz, die als Koroutinen bezeichnet werden.

Eine Koroutine besitzt, im Gegensatz zu einer Funktion, mehrere Ein- und Austrittspunkte mit denen eine unterbrechbare Ausführung realisiert werden kann. Koroutinen werden unterschieden in Semi-Koroutinen und symmetrische Koroutinen. Beim Wechsel zwischen Koroutinen wird der Funktionsaufruf-Stack wiederhergestellt. Dieser Wechsel wird als Kontextwechsel bezeichnet, da sich der Kontext, in dem Anweisungen ausgeführt werden, verändert.

Eine Semi-Koroutine wird von einem sogenannten Generator erzeugt. Dabei handelt es sich um eine Funktion, die Koroutinen erzeugen und deren Fortsetzung steuern kann. Die Koroutine unterbricht die Ausführung mit einer speziellen yield-Anweisung. Daraufhin kehrt die Steuerung zum Generator zurück, der die nächste Koroutine auswählt und fortsetzt.

Die Semi-Koroutine hat den Nachteil, dass die Steuerung immer erst zum Generator zurückkehrt und dessen Funktionsaufruf-Stack wiederherstellt. Der Generator bestimmt im Anschluss die nächste auszuführende Koroutine. Damit gibt es eine unnötige Indirektion in der Fortsetzung des nächsten Prozesses, die einen Einfluss auf die Laufzeit hat.

Symmetrische Koroutinen sind hier effizienter. Sie erlauben den Wechsel von einer Koroutine zu einer beliebigen anderen Koroutine. Der Scheduler kann hier als passive Datenstruktur befragt werden, die lediglich die Koroutinen verwaltet und jeweils die nächste auszuführende Koroutine bestimmt.

[1] wait, reactivate, activate, yield, yield to, fork, terminate, advance, wait until, wait list, reactive list, interrupt, reschedule und resume

Im Allgemeinen ist bei symmetrischen Koroutinen kein Wechsel zwischen Koroutinen möglich, wenn diese jeweils weitere Funktionen aufrufen, in denen sie die Steuerung abgeben. Dabei baut jede Koroutine ihren eigenen Funktionsaufruf-Stack auf, der bei einem Wechsel gesichert und später wiederhergestellt werden muss. Diese Koroutinen bezeichne ich als stack-erhaltende symmetrische Koroutinen. Sie sind für eine prägnante Beschreibung von Prozessen hilfreich, bei denen die Beschreibung auf mehrere Funktionen aufgeteilt werden kann, und bei der aus jeder dieser Funktionen ein Prozesswechsel erfolgen kann.

Die Beschreibung von Stack-erhaltenden symmetrischen Koroutinen erfordert Konzepte zum Zugriff auf den Speicher der Maschine, um so den Stack manipulieren zu können. Außerdem ist eine Sprunganweisung erforderlich. Eine Hinzunahme dieser Konzepte zur Basissprache, um so die Beschreibung durch eine Erweiterung zu ermöglichen, hätte zur Folge, dass prozessorientierte Modelle ebenfalls direkte Speicherzugriffe enthalten würden und eine robuste Speicherverwaltung selbst implementieren müssten. Um die Modelle prägnant und frei von Speicherzugriffsfehlern zu halten, sollte die Basissprache kein Konzept für den Speicherzugriff enthalten. Für die Modellierung hilfreich ist dagegen ein Konzept für den Zugriff auf Objekte mit Referenzen. Ein ähnliches Konzept wird ebenfalls von der Sprache SLX bereitgestellt.

SLX unterscheidet zwei Arten von Werten für Variablen: i) Objektvariablen, die ein Objekt global enthalten, ii) Objektvariablen, die ein Objekt lokal auf dem Stack einer aufgerufenen Funktion enthalten und ii) Zeigervariablen eines bestimmten oder unbestimmten Typs. Obwohl SLX diese Unterscheidung vornimmt, ist mit dem Zeigerkonzept kein direkter Zugriff auf den Speicher möglich. Es dürfen keine arithmetischen Operationen auf den Werten von Zeigervariablen durchgeführt werden. Das Typkonzept von SLX ist durch diese Unterscheidung unnötig kompliziert, da der Modellierer mit dem Problem konfrontiert wird, wie er Werte zu speichern hat. Die Entscheidung für Objekt- und Zeigervariablen in SLX hat vermutlich den Hintergrund eine laufzeiteffiziente Implementierung von Simulationsmodellen zu erlauben. Diese Entscheidung erschwert jedoch auch die Verständlichkeit der Modelle. Um die Modelle prägnant zu halten, nimmt die Basissprache DBL hier eine Vereinfachung vor. In DBL enthält eine Variable entweder eine Objektreferenz oder einen Wert eines primitiven Typs.

Scheduling-Anweisungen

In einem prozessorientierten Modell reagieren Prozesse auf Zeit- und Zustandsereignisse und lösen diese aus. Dazu sind spezielle Anweisungen hilfreich. SLX unterscheidet eine unterbrechbare Zeitfortschrittsanweisung (advance) und eine Anweisung für das Warten auf die Erfülltheit einer Bedingung (wait until). Für die Unterbrechung eines Zeitfortschritts ist eine Anweisung interrupt vorhanden. Ein unterbrochener Prozess lässt sich mit resume oder mit reschedule fortsetzen. Bei einem resume wird der Prozess mit seiner Restzeit im Terminkalender vermerkt. Dagegen ist es bei einem reschedule möglich, die Restzeit neu festzulegen. Diese Anweisungen erfordern die Verwaltung von Prozessen in Datenstrukturen mit einem effizienten Zugriff.

Prozesse, die auf ein Zeitereignis warten, werden in einem Prozessterminkalender verwaltet. Hier ist der schnelle Zugriff auf den Prozess mit der kleinsten Zeit entschei-

dend. Neben einer effizienten Realisierung von Kontextwechseln gehören insbesondere auch die Zugriffe auf den Prozessterminkalender zu den wesentlichen Performancefaktoren in einem prozessorientierten Modell [35].

Die von SLX als Teil der Basissprache bereitgestellten Scheduling-Anweisungen lassen sich in vier Kategorien einteilen: i) Anweisungen für einen unmittelbaren Kontextwechsel, ii) Anweisungen zur Erzeugung weiterer paralleler Prozesse, iii) Anweisungen für den Umgang mit Zeitereignissen und iv) Anweisungen für den Umgang mit Zustandsereignissen.

Ein Kontextwechsel wird unmittelbar vollzogen, sobald ein Prozess eine Warte--Anweisung (wait), eine Anweisung zur Fortsetzung des nächsten Prozesses mit der aktuellen Modellzeit und gleicher Priorität (yield) oder eines bestimmten nächsten Prozesses mit der aktuellen Modellzeit und gleicher Priorität (yield-to) erreicht. Das Aufwecken eines wartenden Prozesses ist mit einer Anweisung für die Reaktivierung (reactivate) möglich. Diese vier Anweisungen stellen eine Umsetzung für das Konzept der stack-erhaltenden symmetrischen Koroutinen unter zusätzlicher Berücksichtigung von Prozessprioritäten und Modellzeit in SLX dar.

Es ist prinzipiell möglich, die weiteren Scheduling-Anweisungen auf der Basis dieser Koroutinen-Anweisungen zu beschreiben. Im Folgenden diskutiere ich die Notwendigkeit, die weiteren Anweisungen in die Basissprache aufzunehmen und die Möglichkeit diese als Erweiterung zu beschreiben. Im Allgemeinen gilt, dass eine Aufnahme in die Basissprache eine laufzeiteffiziente Implementierung erlaubt, während eine Beschreibung als Erweiterung den Umfang der Basissprache reduziert.

3.6.4 Diskussion zu Scheduling-Anweisungen in SLX

SLX unterscheidet im Laufzeitsystem verschiedene Listen, in denen Prozesse verwaltet werden. Es gibt eine Liste Moving Pucks (MP) der Prozesse, die für eine Ausführung zum aktuellen Modellzeitpunkt vorgemerkt sind. Die auf ein zukünftiges Zeitereignis wartenden Prozesse werden in der Liste der Scheduled Pucks (SP) hinterlegt. Prozesse, die auf eine Reaktivierung warten, befinden sich in der Liste der Waiting Pucks (WP). Zusätzlich gibt es eine Liste für unterbrochene Prozesse Interrupted Pucks (IP), sowie eine Liste für Prozesse, die auf ein Zustandsereignis warten Conditioned Waiting Pucks (CWP).

In SLX gibt es allgemein eine 1:n-Zuordnung zwischen einem aktiven Objekt und einer Menge von Prozessen, die auf diesem Objekt operieren. Die verschiedenen Prozessausführungen werden in SLX als Pucks bezeichnet. Der Vorteil dieser Zuordnung sind quasi-parallele Prozesse im Kontext eines aktiven Objektes. Die Alternative ist eine 1:1-Zuordnung, bei der für jeden Prozess ein aktives Objekt erzeugt wird. Soll ein Prozess den Zustand eines aktiven Objektes, das einem anderen Prozess zugeordnet ist, ändern, so ist dies mit einer einfachen Zuweisung möglich. Lediglich der Kontext muss in diesem Fall explizit mit angegeben werden.

Da eine 1:n-Zuordnung lediglich einen vereinfachten Umgang mit Prozesses innerhalb eines Kontextes darstellt, wird dieses Konzept nicht in die Basissprache DBL aufgenommen. Zudem ist eine Beschreibung durch eine Erweiterung möglich. Die Erweiterung muss dazu die Anpassung von Ausdrücken erlaubten, so dass ein anderer Kontext gesetzt werden kann.

Unbestimmtes Warten und Reaktivierung

Das unbestimmte Warten wird mit einer wait-Anweisung ausgelöst. Diese ordnet den aktuellen Prozess am Ende der Prozesse mit gleicher Priorität in der MP-Liste ein. Die Reaktivierung eines wartenden Prozesses erfolgt mit einer reactivate-Anweisung, die den nächsten auszuführenden Prozess aus der MP-Liste entfernt und diesem die Steuerung übergibt.

Eine Beschreibung dieser Anweisungen durch eine Erweiterung erfordert Sprachkonzepte für die Implementierung stack-erhaltender symmetrischer Koroutinen und eine Implementierung der MP-Liste in der Basissprache. Um die Modelle in der Basissprache prägnant und verständlich zu halten, ist eine Beschreibung durch eine Erweiterung, wie in Abschnitt 3.6.3 erläutert, nicht sinnvoll.

Steuerung an einen anderen oder an einen bestimmten Prozess abgeben

Die yield- und yield-to-Anweisung sind sich ähnlich. Beide erlauben die Abgabe der Steuerung an einen vermerkten Prozess zum aktuellen Modellzeitpunkt. Während bei yield der nächste Prozess mit der höchsten Priorität gewählt wird, ist bei yield-to eine genaue Angabe des Prozesses möglich. Da mit beiden Anweisungen auf die MP-Liste zugegriffen werden muss, und diese nicht in der Basissprache vorliegt, ist eine Beschreibung als Erweiterung nicht möglich. Beide Anweisungen sollten daher zur Basissprache dazugehören.

Erstmalige Aktivierung

In SLX liegt mit activate eine Anweisung für die Erzeugung einer weiteren Prozessausführung für ein aktives Objekt vor. Da es mit reactivate bereits eine Anweisung für die Fortsetzung einer Prozessausführung gibt, könnten beide Anweisungen auch zusammengefasst werden. Ersetzt man jedoch activate durch reactivate, so müsste das reactivate nun jedesmal prüfen, ob für das aktive Objekt bereits eine Ausführung vorliegt und wenn nicht, eine entsprechende Ausführung erzeugen. Das diese Prüfung jedesmal erfolgen müsste, hätte sie einen erheblichen Einfluss auf die Laufzeit. Aus diesem Grund ist es besser eine activate-Anweisung in die Basissprache DBL aufzunehmen.

Unbestimmtes Warten und Reaktivieren einer Liste von Prozessen

Die Anweisung wait list beschreibt ein unbestimmtes Warten, bei dem der aktuelle Prozesse in einer angegebenen Liste hinterleg wird. Alle Prozesse in dieser Liste werden bei Aufruf der Anweisung reactivate list wieder aktiviert. Beide Anweisungen lassen sich durch eine Erweiterung beschreiben. Die Anweisung wait list kann auf eine Anweisung für das Hinzufügen des aktuellen Prozesses, gefolgt von der Anweisung wait abgebildet werden. Die Anweisung reactivate list lässt sich auf Anweisungen abbilden, die über die Liste iterieren und für jeden gespeicherten Prozess die Anweisung reactivate aufrufen.

Anweisungen für den Umgang mit Zeitereignissen

Für das Warten auf den Eintritt eines Zeitereignisses steht in prozessorientierten Simulationssprachen eine zeitbedingte Warteanweisung bereit, die unter Angabe einer Zeit d aufgerufen wird. Der aufrufende Prozess wartet, ausgehend von der aktuellen Modellzeit t_i, bis die Modellzeit den Wert $t_j = t_i + d$ mit $t_j > t_i$ erreicht hat.

In SLX gibt es hierfür die Anweisung advance d. Alle Prozesse, die auf den Eintritt der Zeit t_j warten werden entsprechend ihrer Priorität in die MP-Liste einsortiert. Bei gleicher Priorität erfolgt die Sortierung in der Reihenfolge in der die Prozesse die advance-Anweisung aufgerufen haben. Der erste Prozess in der MP-Liste erhält die Steuerung.

Die auf Zeitereignisse wartenden Prozesse werden in einer Min-Heap-Datenstruktur verwaltet, die drei Grundoperationen unterstützen muss: i) das Hinzufügen eines Elements, ii) das Entfernen eines Elementes und iii) die Entnahme des kleinsten enthaltenen Elementes.

Ein Min-Heap kann auf verschiedene Arten implementiert sein. Das Ziel einer Simulationssprache ist die Verwendung einer Min-Heap-Implementierung mit der bestmöglichen Laufzeiteffizienz. Eine naive Implementierung mit schlechter Laufzeiteffizienz stellt die Verwendung einer Liste dar, die nach der Modellzeit der eingetragenen Prozesse sortiert ist. Die Entnahme kann hier zwar mit der Komplexität $O(1)$ erfolgen. Jedoch besitzt das Einfügen eines Prozesses die Komplexität $O(n)$.

Die Laufzeit von Min-Heap-Algorithmen in Simulationen ist in verschiedenen Arbeiten bereits untersucht [68] [18] [76]. Dabei wurde festgestellt, dass es nicht die eine beste Implementierung gibt. Die Laufzeiteffizienz hängt immer von der Verteilung der Zeitereignisse in einem konkreten Modell ab. Je nach Verteilung arbeiten die Algorithmen über den Gesamtlauf einer Simulation effizienter oder weniger effizient.

Ein Min-Heap-Algorithmus, der in der Basissprache DBL implementiert ist, verwendet zur Objektverwaltung die gleichen Objektreferenzen, die auch Modelle in DBL verwenden. Damit wird für diese Objektreferenzen immer auch eine automatische Speicherverwaltung in der Zielsprache C++ bereitgestellt. Der Zugriff auf die Objekte im Prozessterminkalender kann damit weniger effizient sein als eine direkte Implementierung in C++, die Kenntnisse über die Besonderheit des Prozessterminkalenders besitzt und für diesen keine automatische Speicherverwaltung verwendet.

Prozessorientierte Simulationsmodelle realisieren im Allgemeinen viele Zeitereignisse. Da der Einfluss der Implementierung des Prozessterminkalenders damit entscheidend für die Laufzeit der Simulation ist, erfolgt die Bereitstellung einer Zeitverbrauchsanweisung als Basiskonzept in DBL. Die Implementierung kann damit in C++ vorgenommen werden und außerdem bestehende C++-Implementierungen von Min-Heap-Algorithmen wiederverwenden.

Unterbrechung eines zeitbedingten Wartens und Fortsetzung

In SLX kann das Warten eines Prozesses auf ein Zeitereignis mit der Zeit t durch einen anderen Prozess mit einer interrupt-Anweisung, die zu einem Zeitpunkt i mit $i < t$ aufgerufen wird, unterbrochen werden. Die Anweisung resume trägt den Prozess, ausgehend von der aktuellen Zeit $time$, mit der Restzeit $time + t - i$ im Prozessterminkalender ein. Dagegen erlaubt die Anweisung reschedule die Angabe einer

```
X x = new X();

active class X {
    actions {
        advance 10;
    }
}

void main() {
    activate x;
    advance 2;

    // für x verbleiben 8 Zeiteinheiten
    interrupt x;
    advance 2;

    // plant die Fortsetzung von x mit den gemerkten 8 Restzeiteinheiten zum Zeitpunkt 12
    resume x;
    advance 2;

    // plant die Fortsetzung von x mit 4 Zeiteinheiten zum Zeitpunkt 10
    reschedule a with 4;
}
```

Listing 3.5: Basisprogramm

neuen Restzeit. Unterbrochene Prozesse werden in SLX in einer Liste Interrupted Pucks (IP-Liste) verwaltet. Listing 3.5 zeigt ein Beispiel.

Die Unterbrechung eines zeitbedingten Wartens wird für die Abbildung von Ausnahmesituationen in einem modellierten System verwendet. Das kann z.B. der Ausfall einer Maschine sein, die für eine bestimmte Reparaturzeit nicht mehr zur Verfügung steht.

Eine Unterbrechungsanweisung, die in der Basissprache implementiert ist, lässt sich nur dann laufzeiteffizient realisieren, wenn der Prozessterminkalender ebenfalls in der Basissprache vorliegt, so dass der Zugriff auf diese Datenstruktur möglich ist. Wenn der Prozessterminkalender jedoch in der Zielsprache implementiert ist, so kann die Unterbrechung eines Prozesses nur durch eine spezielle Anweisung iadvance für das Warten auf Zeitereignisse realisiert werden. Die Anweisung iadvance muss in allen Fällen, in denen ein Prozess, der auf ein Zeitereignis wartet und unterbrechbar sein soll, an Stelle der zeitbedingten Warteanweisung advance der Basissprache verwendet werden.

Listing 3.6 zeigt das Beispiel aus Listing 3.5 unter Verwendung einer unterbrechbaren Anweisung iadvance, die in der Basissprache implementiert ist. Durch eine Spracherweiterung lassen sich die notwendigen Anweisungen für den Aufruf und die Behandlung einer Unterbrechung auf jeweils eine Anweisung reduzieren.

```
X x = new X();

active class InterruptableAdvancer {
    int i;
    Object caller;
```

```
6       boolean interrupted = false;
7       double startTime;
8
9       new(int t, Object caller) {
10          self.t = t;
11          self.caller = caller;
12      }
13
14      actions {
15          startTime = time;
16          advance t;
17          if (!interrupted) {
18              caller.finished = true;
19              reactivate caller;
20          }
21      }
22
23      double getRemainingTime() {
24          return time - startTime;
25      }
26
27      void deactivate() {
28          interrupted = true;
29      }
30  }
31
32  active class X {
33      boolean interrupted = false;
34      boolean finished = false;
35      double rescheduleTime = 0;
36      boolean resumedOrRescheduled = false;
37
38      actions {
39          InterruptableAdvancer advancer = new InterruptableAdvancer(10, self);
40          activate advancer;
41
42          // jede Erweiterungsinstanz initialisiert den Zustand neu
43          interrupted = false;
44          finished = false;
45          rescheduleTime = 0;
46          resumedOrRescheduled = false;
47
48          while (!finished) {
49              wait;
50
51              while (interrupted) {
52                  advancer.deactivate();
53                  if (resumedOrRescheduled) {
54                      if (rescheduleTime == 0) {
55                          rescheduleTime = advancer.getRemainingTime();
56                      }
57                      advancer = new InterruptableAdvancer(rescheduleTime, self);
58                      activate advancer;
59                      resumedOrRescheduled = false;
60                      interrupted = false;
61                  } else {
62                      wait;
```

```
63                  }
64              }
65          }
66      }
67  }
68
69  void main() {
70      activate x;
71      advance 2;
72
73      x.interrupted = true;
74      reactivate x;
75      advance 2;
76
77      x.resumedOrRescheduled = true;
78      reactivate x;
79      advance 2;
80
81      x.resumedOrRescheduled = true;
82      x.rescheduleTime = 4;
83      reactivate x;
84  }
```

Listing 3.6: Erweiterung

Die Implementierung erfolgt mit einem Objekt einer aktiven Klasse InterruptableAdvancer, das parallel zum Aufrufer ausgeführt wird und auf den Eintritt des Zeitereignisses wartet. Der Aufrufer ist nun durch andere Prozesse reaktivierbar. Sobald eine Reaktivierung erfolgt, wird festgestellt, ob es sich um eine Unterbrechung handelt oder ob das Zeitereignis regulär eingetreten ist.

Die notwendige Erzeugung eines aktiven Objektes InterruptableAdvancer beeinflusst die Laufzeit negativ. Eine möglichst laufzeiteffiziente Realisierung der Unterbrechung von zeitbedingt wartenden Prozessen sollte daher in der Zielsprache vorgenommen werden.

Bedingtes Warten

Das Warten auf den Eintritt eines Zustandsereignisses lässt sich in SLX prägnant durch die Angabe einer Bedingung in einer wait-until-Anweisung beschreiben. Ein Prozess wird in einen Wartezustand versetzt und automatisch reaktiviert, sobald sich der Wert einer Variablen, die in der Bedingung vorkommt, verändert. Ist die Bedingung erfüllt, wird der Prozess fortgesetzt. Im anderen Fall, wartet der Prozess bis erneut auf eine Änderung.

Um die Überprüfung der Bedingung nicht bei jeder Variablenänderung durchführen zu müssen, erlaubt es SLX, bestimmte Variablen als sogenannte Kontrollvariablen zu deklarieren. Nur wenn sich der Wert einer Kontrollvariablen in der Bedingung einer wait-until-Anweisung ändert, wird der Prozess reaktiviert. Kontrollvariablen erlauben damit eine effizientere Behandlung von Zustandsereignissen.

Das Konzept für das Warten auf Zustandsereignisse besteht aus der wait-until-Anweisung, aus Kontroll-Variablen und aus einer speziellen Zuweisung. Die wait-until-Anweisung lässt sich einfach als Erweiterung einer Anweisung beschreiben, die auf

eine wait-Anweisung abgebildet wird. Eine Kontroll-Variable lässt sich ebenfalls als Erweiterung einer Variablen festlegen. Diese wird auf eine Basisvariable und eine Datenstruktur für Prozesse, die auf eine Änderung an der Variablen warten, abgebildet.

Schwierig ist dagegen die Beschreibung der Zuweisung durch eine Erweiterung. Erfolgt die Zuweisung an eine Kontrollvariable, so müssen wartende Prozesse reaktiviert werden. Dies ließe sich mit einem Konzept für eine Operatorüberladung so umsetzen, dass das bestehende Konzept für die Zuweisung auch für Kontrollvariablen verwendet werden kann, und sich bei Kontrollvariablen anders verhält. Eine Konzept für die Operatorüberladung ist jedoch aufgrund seiner Komplexität in DBL nicht enthalten.

Eine spezielle Zuweisung kann in DBL nur über eine neue Anweisung als Erweiterung einer Basiszuweisung festgelegt werden. Dabei ist es notwendig, ein Schlüsselwort einzuführen, so dass sich die neue Zuweisung von der bestehenden Basiszuweisung syntaktisch unterscheidet. Bei dieser Lösung muss der Modellierer immer die passende Zuweisung benutzen. Verwendet er dagegen die Basiszuweisung für eine Kontrollvariable, so werden die wartenden Prozesse nicht reaktiviert.

Eine Modellbeschreibung ist prägnanter und weniger fehleranfällig, wenn die erforderliche spezielle Zuweisung an Kontrollvariablen zur Basissprache dazugehört und sich von der Basiszuweisung nicht unterscheidet. Der Zielsprachen-Compiler stellt fest, ob eine Zuweisung an eine Kontrollvariable erfolgt, und erzeugt zusätzliche Anweisungen für die Behandlung. Damit sind die wait-until-Anweisung und Kontrollvariablen ebenfalls Teil der Basissprache.

3.6.5 Vereinfachungen

Wiederkehrende Ereignisse, die nur durch das Ende der Simulation begrenzt sind, werden in einem prozessorientierten Modell mit einer Endlosschleife beschrieben, die Teil einer Prozesslebenslaufbeschreibung ist. In einer Programmiersprache verwendet man eine while-Schleife, deren Bedingung true und damit immer erfüllt ist. In SLX steht mit der forever-Anweisung eine Schleifen-Anweisung bereit, die einer solchen while-Schleife entspricht. Die forever-Anweisung lässt sich durch eine Erweiterung einer Anweisung leicht beschreiben und muss daher nicht zur Basissprache dazugehören.

3.6.6 Parallelität im Kontext eines aktiven Objektes

Eine Besonderheit von SLX stellt ein Beschreibungsmittel für quasi-parallele Ausführungen innerhalb des Kontextes eines aktiven Objektes dar. Hierfür steht eine spezielle fork-Anweisung bereit, die weitere Anweisungen für eine parallele Ausführung enthält. Erreicht die Ausführung die fork-Anweisung, so wird eine weitere parallele Ausführung erstellt, die in der ersten Anweisung innerhalb des fork beginnt. Die ursprüngliche Ausführung, die das fork erreicht hat, führt nur die Anweisungen nach der fork-Anweisung aus.

Diese lokale Form von Parallelität lässt sich durch die Erzeugung eines aktiven Objektes nachbilden. Der Prozesslebenslauf dieses neuen Objektes enthält die Anweisungen, die innerhalb eines SLX-fork stehen. Da sich der Kontext, in dem die Anweisungen ursprünglich definiert wurden, nun aber verändert hat, ist es erforderlich die Ausdrücke durch Hinzufügen eines Verweises auf das Objekt, in dem die Anweisungen ursprünglich definiert waren, anzupassen. Der Erweiterungsmechanismus muss dazu

die Veränderung des Basisprogramms unterstützen. Dies kann durch den Zugriff auf den abstrakten Syntaxbaum eines Basisprogramms innerhalb der Semantikbeschreibung einer Erweiterung erreicht werden.

3.6.7 Ausgewählte Konzepte in DBL

DBL enthält die folgenden Scheduling-Anweisungen: activate, wait, reactivate, yield, yield-to, advance und wait until mit Kontrollvariablen. Die Scheduling-Anweisungen interrupt, resume und reschedule sind aus Gründen der Vereinfachung nicht Teil der prototypischen Implementierung der Basissprache DBL. Die Anweisungen wait list und reactivate list sind als Erweiterungen beschrieben.

Zusätzlich gibt es in DBL eine 1:1-Zuordnung zwischen einem aktiven Objekt und einer Prozessausführung. Ein aktives Objekt ist ein Objekt einer aktiven Klasse. In DBL gibt es dazu das Schlüsselwort active, das einer Klassendefinition vorangestellt wird. Aktive Objekte besitzen außerdem eine Priorität, die standardmäßig 1 ist. Der Programmeintrittspunkt der main-Funktion stellt ebenfalls eine Prozessausführung dar, deren Priorität `0` beträgt. Wenn die main-Funktion endet, so endet auch die Simulation inklusive der wartenden Prozesse. Damit sind die Anweisungen fork und terminate aus SLX in DBL nicht notwendig.

Die Semantik der DBL-Konzepte für die prozessorientierte Modellierung entspricht der Semantik von SLX. Die Semantik der objektorientierten Konzepte und der Typen entspricht dagegen der Semantik von Java. Da die vorliegende Arbeit eine Abbildung von DBL in die Sprachen Java und C++ vornimmt und insbesondere eine neuartige Methode für die Implementierung hochlaufzeiteffizienter Kontextwechsel in C++ beschreibt, ist es notwendig, die Semantik der prozessorientierten Konzepte von DBL präzise festzulegen. Dazu gebe ich die erforderlichen Laufzeitstrukturen als eine Ergänzung zum DBL-Metamodell an und beschreibe deren Semantik.

Abbildung 3.9 zeigt die Laufzeitstukturen und ihre Beziehungen zu statischen Strukturen, die durch das DBL-Metamodell definiert sind. Jedes ausgeführte DBL-Modell besitzt einen Scheduler, der die aktiven Objekte in Abhängigkeit ihres Laufzeitzustandes in verschiedenen Listen verwaltet und das nächste auszuführende aktive Objekt bestimmt.

Für jede aktive Klasse kann es viele aktive Objekte geben, deren Laufzeitstruktur durch die Klasse ActiveObject beschrieben wird. Ein aktives Objekt besitzt eine aktuelle Ausführungsposition pc, die auf eine Anweisung im actions-Teil der zugehörigen aktiven Klasse verweist. Weitere Laufzeitattribute sind eine optionale zukünftige Modellzeit moveTime, auf deren Eintritt das aktive Objekte wartet, eine Priorität priority und eine Beschreibung des Laufzeitzustandes state. Der Wert des Laufzeitzustandes state entspricht einem der in Abbildung 3.10 dargestellten Zustände. Diese beschreibt die Übergänge zwischen Laufzeitzuständen beim Aufruf von Scheduling-Anweisungen und bei der Auswahl durch den Scheduler. Die Attribute des Laufzeitzustandes sind auch innerhalb eines DBL-Modells lesend zugreifbar.

Zu Beginn gibt es nur ein aktives Objekt, das für die main-Funktion erstellt wird. Diese kann nun weitere aktive Objekte erzeugen. Nach der Instanziierung einer aktiven Klasse befindet sich das betreffende aktive Objekt zunächst im Zustand created. Erst nachdem eine activate-Anweisung auf einem aktiven Objekt aufgerufen wird, ist

dieses für die Ausführung bereit und wird am Ende der Liste ml (moving list) eingefügt. Beim Aufruf von activate kann außerdem eine Priorität angegeben werden.

Es gibt ein ausgezeichnetes aktives Objekt current. Alle aktiven Objekt mit der gleichen Modellzeit, wie das aktuelle aktive Objekt, befinden sich in der Liste ml. Dabei gilt: moveTime = time. Die aktiven Objekte sind in der Liste ml nach ihrer Priorität sortiert. Für aktive Objekte mit der gleichen Priorität erfolgt die Sortierung nach der Reihenfolge der Einsortierung in den Listen ml bzw. sl. Sobald das aktuelle aktive Objekt eine Scheduling-Anweisung aufruft, die zur Einsortierung des Objektes in einer der Listen führt, wird das nächste aktive Objekt aus der Liste ml entnommen und dieses zum neuen aktiven Objekt current.

In der Liste sl (scheduled list) befinden sich aktive Objekte, die auf ein Zeitereignis warten und deren Modellzeit somit größer als der aktuelle Fortschritt der Modellzeit ist. Es gilt: moveTime > time. Ein aktives Objekt wird nur durch die Ausführung einer advance-Anweisung in der Liste sl hinterlegt. Wenn die Liste ml keine aktiven Objekt mehr enthält, so werden alle aktiven Objekte mit der kleinsten Modellzeit aus der Liste sl entfernt und in die Liste ml geschrieben. Die aktuelle Modellzeit time wird auf die Modellzeit der entnommenen aktiven Objekte gesetzt.

Aktive Objekte, die unbestimmt auf eine Reaktivierung warten, werden in der Liste wl (waiting list) verwaltet. Ein aktives Objekt wird nur durch den Aufruf einer wait-Anweisung in der Liste wl hinterlegt. Sobald ein anderes aktives Objekt eine reactivate-Anweisung auf einem hinterlegten aktiven Objekt aufruft, wird das hinterlegte aktive Objekt aus der Liste wl entfernt und am Ende der Liste ml hinzugefügt.

Beim Aufruf einer wait-until-Anweisung wartet ein aktives Objekt auf den Eintritt eines Zustandsereignisses, das durch eine Bedingung beschrieben ist. Falls die Bedingung beim Aufruf erfüllt ist, wird die nachfolgende Anweisung ausgeführt. Falls die Bedingung nicht erfüllt ist, wird das aktive Objekt in der Liste waiting in jeder Kontrollvariablen, die in der Bedingung verwendet wird, eingefügt. Ändert sich der Wert einer der Kontrollvariablen, so werden alle aktiven Objekten in der Liste waiting reaktiviert, so dass die Bedingung erneut geprüft wird. Falls die Bedingung erfüllt ist, werden die aktiven Objekte aus der Liste waiting entfernt.

Listing 3.7 zeigt ein Beispielmodell in DBL. Zunächst werden x und y als globale aktive Objekte erzeugt und befinden sich im Zustand created. Nach deren Aktivierung befinden sich beide Objekt im Zustand ready und sind in der Liste ml entsprechend ihrer Priorität einsortiert. Das aktive Objekt der main-Funktion wird nach Aufruf von advance in der Liste sl mit der moveTime = 1 hinterlegt. Danach wird das aktive Objekt y zum aktuellen aktiven Objekt. Die wait-until-Anweisung von y ist nicht erfüllt und das Objekt wird in der Liste waiting der Kontrollvariablen i eingefügt. Im Anschluss wird das Objekt x zum aktuellen Objekt. Dieses greift auf seinen eigenen Laufzeitzustand und die Modellzeit zu und wartet danach mit der wait-Anweisung unbestimmt. Die Liste ml ist leer und es wird die kleinste Zeit in der Liste sl bestimmt. Das Attribut time im Scheduler wird auf 1 gesetzt und das aktive Objekt der main-Funktion erhält die Steuerung. Dieses ändert die Kontrollvariable i auf 2. Damit wird das aktive Objekt y reaktiviert und die Liste ml eingefügt. Das aktive Objekt der main-Funktion wird nach Aufruf der yield-Anweisung am Ende der Liste ml eingefügt und das am Anfang der Liste ml befindliche Objekt y erhält die Steuerung. Dieses prüft die Bedingung i > 0 und stellt die Erfülltheit fest. Danach reaktiviert das Objekt y das

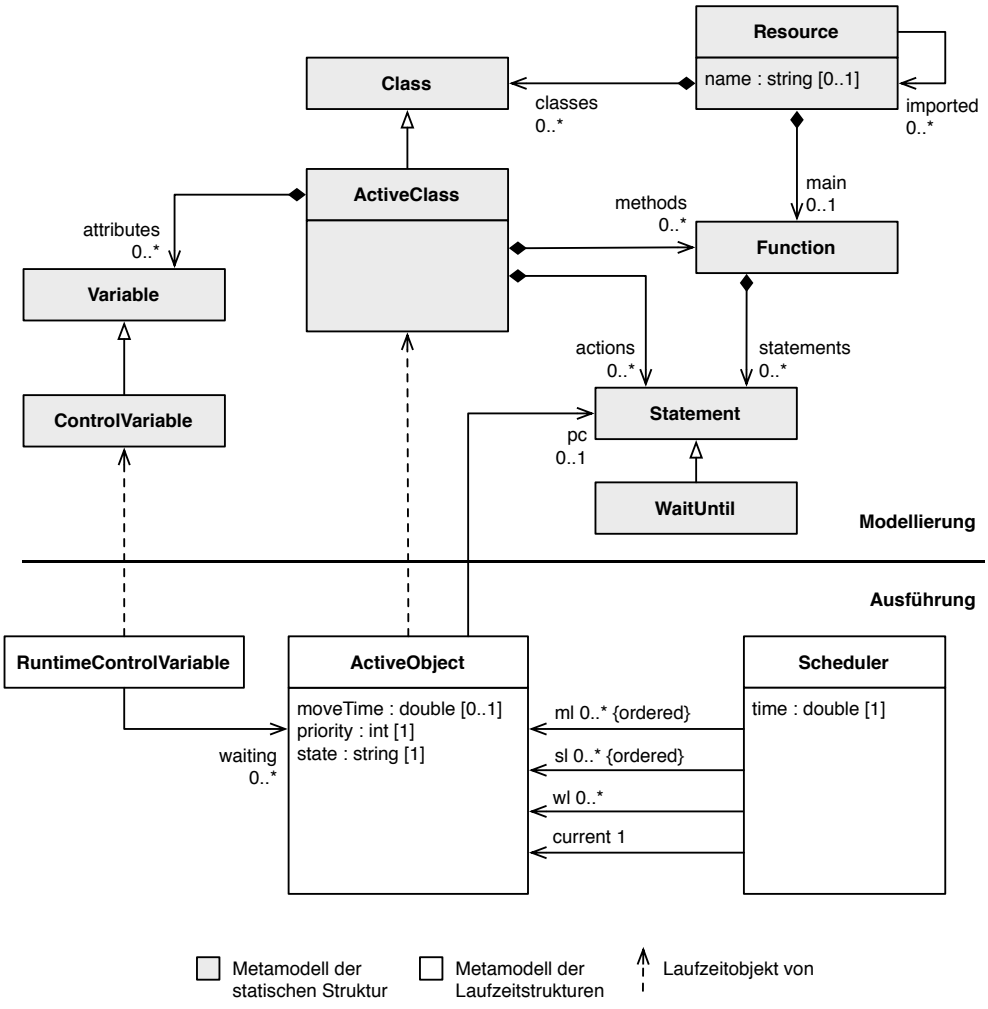

Abbildung 3.9: Beschreibung der möglichen Laufzeitstrukturen in einem ausgeführten DBL-Modell.

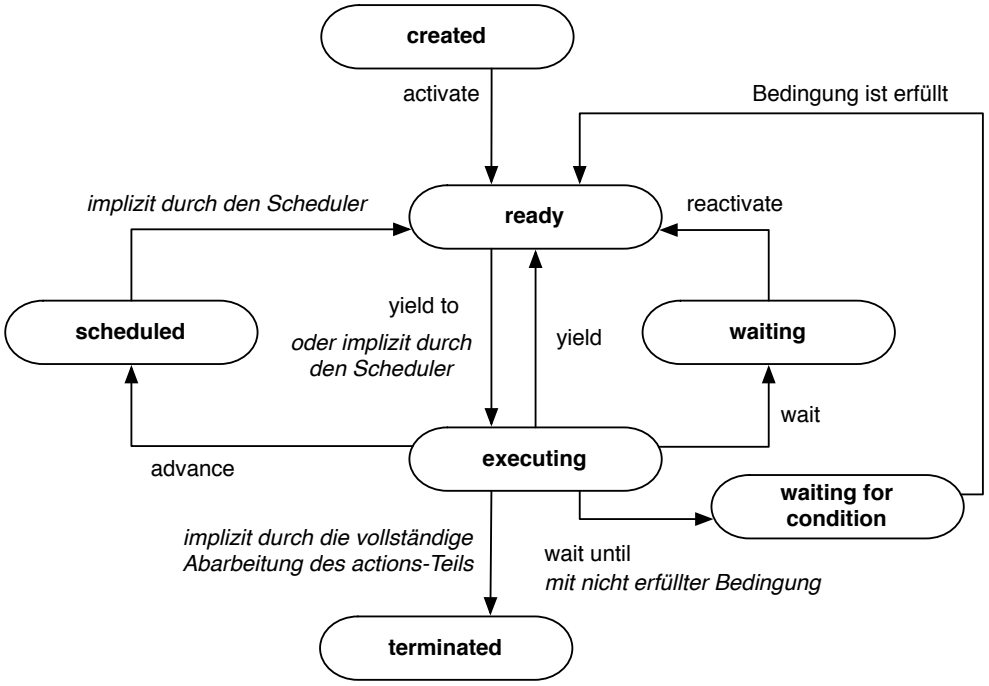

Abbildung 3.10: Die Laufzeitzustände eines aktiven Objektes in DBL.

Objekt x, das am Ende der Liste ml eingefügt wird. Die Ausführung des Objektes y endet und der Zustand wechselt zu terminated. Das aktive Objekt der main-Funktion erhält wieder die Steuerung. Durch den Aufruf der yield-to-Anweisung wechselt die Steuerung zum Objekt x und das aktive Objekt der main-Funktion wird am Ende der Liste ml eingefügt. Zunächst endet dann die Ausführung von x und die main-Funktion erhält wieder die Steuerung. Deren Ausführung endet dann ebenfalls und damit die Ausführung des Programms.

```
1  X x = new X();
2  Y y = new Y();
3  control int i = 0;
4
5  active class X {
6      actions {
7          int p = self.priority;
8          string s = self.state;
9          double mt = self.moveTime;
10         double t = time;
11         wait;
12     }
13 }
14
15 active class Y {
16     actions {
17         wait until i > 0;
18         reactivate x;
```

```
19      }
20  }
21
22  void main() {
23      activate x;
24      activate y priority 2;
25      advance 1;
26      i = 2;
27      yield
28      yield to x;
29  }
```

Listing 3.7: Beispielprogramm mit prozessorientierten Konzepten

3.7 Anbindung an bestehende Bibliotheken in C++ und in Java

DBL-Modelle werden auf Programme in Java und C++ abgebildet, für die es bereits ein großes Repertoire an Bibliotheken mit vielen bestehenden Funktionen gibt. Die Verwendung von Bibliotheken der Plattform des Zielprogramm soll für DBL ermöglicht werden, um so den Aufwand für eine erneute Implementierung in DBL zu vermeiden und außerdem besonders effizient implementierte Funktionen wiederverwenden zu können. Auf diese Weise lassen sich z.B. abstrakte Datentypen wie Listen und Maps einbinden und müssen nicht erneut in DBL programmiert werden. Des Weiteren können auch bestehende und vor allem effizient implementierte Zufallszahlengeneratoren, die in Simulationssprachen erforderlich sind, verwendet werden.

DBL enthält mit dem Interface-Konzept bereits eine Möglichkeit die Schnittstelle einer Klasse ohne die Angabe einer Implementierung zu definieren. Dieses Konzept kann verwendet werden, um die Schnittstelle für Klassen, die in Java oder in C++ implementiert sind, in DBL festzulegen. Bei der Verwendung eines extern implementierten Interface in einem DBL-Modell, erzeugt der Zielsprachen-Compiler einen Zugriff auf die entsprechende Klasse oder auf ein Interface in einer Bibliothek der Zielsprache und generiert in diesem Fall keinen Ziel-Code für das DBL-Interface.

Ein DBL-Modell verwendet für eine bestimmte Datenstruktur ein DBL-Interface mit einer externen Implementierung. Auf diese Weise kann ein Modell immer die gleiche Schnittstelle für eine extern implementierte Funktion benutzen. Im Allgemeinen wird sich dabei die Benennung einer Datenstruktur in Java und in C++ unterscheiden. Außerdem besitzen beide Sprachen verschachtelte Namensräume. Diese Probleme lassen sich durch die Angabe jeweils einer Klasse oder eines Interface inklusive ihres vollständigen Namensraumes für jede Zielsprache lösen. DBL erlaubt dazu die Definition eines Bindings innerhalb eines DBL-Interface.

Listing 3.8 zeigt ein Beispiel für ein extern implementiertes Interface List, das zwei Bindings definiert. Die Methoden der Datenstruktur stammen vom Java-Interface List. Für die Abbildung nach C++ wird ein entsprechendes Interface eingeführt.

```
1  interface List {
2    bindings {
3      "java" -> "java.util.List"
```

```
4         "c++" -> "dbl::List"
5   }
6
7   boolean add(Object e);
8   boolean add(int index, Object e);
9   void clear();
10  boolean contains(Object e);
11  int size();
12  Object get(int index);
13  int indexOf(Object e);
14  boolean isEmpty();
15  boolean remove(Object e);
16  Object set(int index, Object e);
17  Object array[] toArray();
18  Iterator iterator();
19 }
```

Listing 3.8: Binding für eine Liste als extern implementierte Datenstruktur.

Die Angabe eines Bindings auf eine Klasse in der Zielsprache erfolgt analog. In Java besitzt das Interface List verschiedene konkrete Implementierungen wie z.B. eine ArrayList und eine LinkedList. Eine dieser Implementierungen wird bei der Erzeugung der Datenstruktur ausgewählt. Listing 3.9 zeigt die Definition eines Bindings für eine ArrayList.

Während für Java die Klasse ArrayList direkt verwendet werden kann, ist für C++ die Einführung einer Zwischenklasse für eine Weiterleitung der Aufrufe, entsprechend des List-Interface, an die entsprechende Datenstruktur vector aus der C++-Standard-Template-Library (STL) erforderlich. Diese Zwischenklasse hat dabei keine negativen Auswirkungen auf die Laufzeiteffizienz des C++-Zielprogramms, da ein automatisch optimierender C++-Compiler die entsprechenden Funktionsaufrufe so ersetzt, das die Funktionen der STL-Klasse vector direkt aufgerufen werden.

```
1 interface ArrayList extends List {
2   bindings {
3     "java" -> "java.util.ArrayList"
4         "c++" -> "dbl::ArrayList"
5   }
6 }
```

Listing 3.9: Binding für eine ArrayList als extern implementierte Datenstruktur.

3.7.1 Umgang mit Typ-Parametern der Zielsprache

Die Datenstrukturen der Standardbibliotheken von Java und C++ definieren für die verwalteten Elemente Typ-Parameter, die bei der Abbildung in die Zielsprache festgelegt werden müssen. In Java ist die Angabe dabei optional. In diesem Fall sind die Elemente immer vom Typ Object, der eine implizite Oberklasse für jede Klasse in Java darstellt. Für einen vector aus der C++-STL muss dagegen der konkrete Typ immer angegeben werden. Um eine Abbildung für eine einheitliche Verwendung mit einem DBL-Interface zu erlauben, wird für DBL ein implizites externes Ober-Interface Object eingeführt, das jede DBL-Klasse implementiert.

Für die Ablage und die Entnahme von Elementen ist ein extern implementiertes DBL-Interface Object, das keine Methoden enthält und auf ein Java-Object und eine C++-Klasse Object abgebildet wird, bereits ausreichend. Damit DBL-Objekte auch inhaltlich verglichen werden können, ist die Einführung der Methoden equals und hashCode hilfreich. Außerdem erlaubt die Methode toString eine Darstellung eines Objektes als Zeichenkette.

Anbindung an Java

Die Anbindung an Java erfolgt durch die Definition eines extern implementierten DBL-Interface mit dem Namen Object, das in Listing 3.10 gezeigt wird. Für die Methoden equals und hashCode ist eine Definition in DBL unkompliziert, da boolean und int zu den primitiven Typen von DBL gehören. Die Methode toString definiert als Rückgabe jedoch den Typ String, der in Java durch eine Klasse definiert ist und in DBL lediglich als primitiver Typ vorliegt. Damit ein string-Wert in DBL wie ein String-Objekt behandelt werden kann, gibt es eine Anbindung an die Klasse String. Soll ein Wert vom primitiven Typ string als Objekt verwendet werden, so muss dazu erst ein String-Objekt erstellt werden.

```
1  module stdlib;
2
3  interface Object {
4      bindings {
5          "java" -> "java.lang.Object",
6          "c++" -> "dbl::Object"
7      }
8      boolean equals(Object other);
9      int hashCode();
10     String toString();
11 }
12
13 interface String {
14     bindings {
15         "java" -> "java.lang.String"
16     }
17     String new(string original);
18     int length();
19 }
20
21 void useString() {
22     string s = "value";
23     String os = new String(s);
24     int sl = os.length();
25 }
```

Listing 3.10: Anbindung der Klassen Object und String aus Java.

3.7.2 Anbindung an C++ und C++-STL

Damit die in DBL definierte Klasse ArrayList auch bei einer Abbildung nach C++ eingesetzt werden kann, ist die Definition einer Zwischenklasse ArrayList in C++ erforderlich, die alle Aufrufe an einen vector aus der C++-STL weiterleitet. Ein vector

```
1  class ArrayList : public List {
2  private:
3      std::vector<boost::intrusive_ptr<Object> > vector;
4
5  public:
6      void add(boost::intrusive_ptr<Object> object) {
7          vector.push_back(object);
8      }
9
10     boost::intrusive_ptr<Object> get(int i) {
11         return vector[i];
12     }
13
14     int size() {
15         return vector.size();
16     }
17
18     virtual boost::intrusive_ptr<Iterator> iterator() {
19         return new ArrayListIterator(vector);
20     }
21 };
```

Listing 3.11: C++-Zwischenklasse für ArrayList.

in C++ entspricht dabei semantisch einer ArrayList aus Java. Listing 3.11 zeigt die entsprechende C++-Zwischenklasse ArrayList.

Die Klasse ArrayList erfordert außerdem die Einführung weiterer C++-Klassen für List, Iterator, ArrayListIterator und Object. Diese sind ausschnittsweise in Listing 3.12 dargestellt und befinden sich vollständig im Anhang in Listing A.1. Die Referenzsemantik von DBL wird dabei auf ein Konzept mit dem Namen Intrusive Pointer aus der C++-Boost-Bibliothek abgebildet.

Ein Intrusive Pointer gibt ein referenziertes Objekte automatisch frei sobald der Referenzzähler, der sich im Objekt selbst befindet, den Wert Null erreicht, und das Objekt somit von keinem Intrusive Pointer mehr referenziert wird. Das Konzept des Intrusive Pointer ist eine Möglichkeit eine automatische Speicherverwaltung für C++-Objekte zu implementieren. Ein anderes Konzept ist der Shared Pointer, der Teil von C++11 ist. Beim Einsatz eines Shared Pointers kann der Zeiger this auf das Objekt selbst jedoch nicht mehr innerhalb eines Konstruktors als Shared Pointer für Verweise in anderen Objekten bereitgestellt werden.

Die in C++ erforderliche Zwischenklasse für eine ArrayList kann vermieden werden, in dem die C++-Klasse vector in DBL als externes Interface definiert wird. Ein DBL-Modell, das auf eine besonders effiziente Implementierung von Datenstrukturen angewiesen ist, kann diese als zusätzliche externe Interfaces einführen. Die Anbindung von Java-Datenstrukturen ist lediglich für die Anbindung Java-basierter Tools und für die Übersetzung von DBL-Erweiterungen nach DBL erforderlich.

```
1  class List : public Object {
2  public:
3      virtual void add(boost::intrusive_ptr<Object> object) = 0;
4      virtual boost::intrusive_ptr<Object> get(int i) = 0;
5      virtual int size() = 0;
6      virtual boost::intrusive_ptr<Iterator> iterator() = 0;
7  };
8
9  class Object : public boost::intrusive_ref_counter<Object,
10     boost::thread_unsafe_counter> {
11 public:
12     Object() {}
13     virtual ~Object() {}
14
15     virtual boost::intrusive_ptr<String> toString() const;
16
17     virtual bool equals(boost::intrusive_ptr<Object> other) const {
18         return this == other.get();
19     }
20
21     virtual int hashCode() const {
22         return reinterpret_cast<uintptr_t>(this);
23     }
24 };
```

Listing 3.12: C++-Zwischenklasse für List und Object.

4 Laufzeiteffiziente Prozesskontextwechsel in C++

Die Laufzeit eines Simulators ist ein bedeutender Faktor bei der Wahl einer Simulationssprache. Für eine Simulation kann entscheidend sein, ob sie nach einer Stunde oder erst nach einem Tag ein Ergebnis liefert. Die Laufzeit hängt von vielen Faktoren ab (siehe Abschnitt 2.5). Dazu gehören verschiedene Algorithmen, wie z.B. Scheduling-Algorithmen für die Verwaltung und für den Zugriff auf quasi-parallele Prozesse während einer Simulation, aber auch Algorithmen für die Erzeugung von Zufallszahlen und für Datenstrukturen im Allgemeinen. Diese Algorithmen sind bereits umfangreich untersucht.

Im vorliegenden Kapitel konzentriere ich mich auf die Kontextwechsel zwischen quasi-parallelen Prozessen. Ich beschreibe eine neue Methode für die Abbildung von Prozessdefinitionen auf ein C++-Programm, bei dessen Ausführung Kontextwechsel zwischen Prozessen während der Simulation besonders laufzeiteffizient realisiert werden. Die Methode bildet sämtliche Verhaltensbeschreibungen von Prozessdefinitionen auf eine einzige C++-Funktion ab und realisiert Kontextwechsel nur innerhalb dieser Funktion, in dem die Konzepte *Labels as Values* und *Switch*, zusammen mit einer Emulation von Funktionen, eingesetzt werden. Die Methode wird als *Goto-based Process Context Switching* (GotoSwitching) bezeichnet.

GotoSwitching kann nur als Teil eines automatisierten Übersetzungsprozesses eingesetzt werden, da die Abbildung auf eine einzige Funktion und die notwendige Emulation von Funktionen dazu führen, dass Prozessdefinitionen und Funktionen in C++ nicht mehr prägnant dargestellt werden und der Programmfluss sowie die Zugriffe auf Variablen nur schwer nachvollziehbar sind. Die Methode kann als Grundlage für eine Abbildung von DBL nach C++ verwendet werden.

GotoSwitching ist allgemein für objekt- und prozessorientierte Simulationssprachen anwendbar und nicht auf DBL beschränkt. Die Herausforderung besteht darin, die Methode allgemein zu beschreiben und durch Laufzeitvergleiche mit SLX und anderen Implementierungen für Prozesskontextwechsel nachzuweisen, dass die Methode besonders laufzeiteffiziente Simulationen erlaubt.

Dabei stellt besonders das Erreichen der Laufzeit eines vergleichbaren SLX-Programms eine Herausforderung dar. Der SLX-Compiler übersetzt ein SLX-Programm in ein Assembler-Programm und kann die Ausführung von Prozesskontextwechseln mit Assembler-Instruktionen besonders gut realisieren. Die Laufzeituntersuchungen zeigen, dass Prozesskontextwechsel in SLX-Programmen sehr schnell ausgeführt werden und GotoSwitching sogar noch etwas schneller ist. Der SLX-Compiler implementiert dabei keine automatische Code-Optimierung. Diese Optimierungen sind eine Stärke von C++-Compilern, von der die vorgestellte Methode profitiert.

4.1 Einleitung

Die Ausführung eines prozessorientierten Simulationsmodells erfordert häufige Wechsel von einem aktuell ausgeführten Prozess zum nächsten auszuführenden Prozess. Ein Prozesskontextwechsel tritt auf, sobald ein Prozess aufgrund seiner Prozessdefinition suspendiert werden muss, bis ein bestimmtes Ereignis vorliegt. Die Zeit für die Ausführung eines Kontextwechsels sollte möglichst gering sein, um die Gesamtlaufzeit eines Simulationsmodells gering zu halten.

Der Prozesskontext, der bei einem Wechsel von einem aktuellen Prozess zu einem anderen Prozess gesichert und wiederhergestellt werden muss, besteht aus der aktuellen Ausführungsposition und den aufgerufenen Funktionen. Vor einem Kontextwechsel wird der nächste auszuführende Prozess mit Hilfe eines Scheduling-Algorithmus bestimmt. Nachdem der neue Prozess ermittelt ist, wird der Kontextwechsel durchgeführt.

Ein Prozesskontextwechsel besteht aus den folgenden Schritten, die in C++ zu realisieren sind:

1. der Sicherung der Ausführungsposition des aktuellen Prozesses,
2. der Wiederherstellung der Ausführungsposition des nächsten Prozesses,
3. der Sicherung der Funktionsaufrufe des aktuellen Prozesses und
4. der Wiederherstellung der Funktionsaufrufe des nächsten Prozesses.

Die Funktionsaufrufe eines Programms werden allgemein mit einer Stack-Datenstruktur verwaltet, die als Call-Stack bezeichnet wird. Der für die Ausführung relevante Zustand jeder aufgerufenen Funktion wird auf dem Call-Stack als Stack-Frame gespeichert.

Listing 4.1 zeigt ein DBL-Modell, das eine Prozessdefinition als aktive Klasse `Counter` und eine `main`-Funktion enthält, die die Erzeugung von zwei Prozessen als aktive Objekte festlegt. Die `Counter`-Prozesse verringern wechselseitig den Wert einer globalen Variablen `count` um den Wert 1, solange dieser größer als `0` ist. Die Simulation endet sobald die Variable `count` den Wert `0` erreicht hat.

C++ bietet mit den Konzepten *Labels as Values* und *Switch* eine besonders laufzeiteffiziente Möglichkeit innerhalb eines Programms, den Programmfluss dynamisch zu verzweigen. Damit lässt sich bereits der Teil eines Prozesskontextwechsels realisieren, der die Ausführungsposition sichert und wiederherstellt. Die Sicherung und Wiederherstellung der aufgerufenen Funktionen ist mit diesen Konzepten jedoch nicht möglich und stellt eine Herausforderung dar.

4.1.1 Wechsel der Ausführungsposition mit *Labels as Values*

Das Konzept *Labels as Values*[1] erlaubt einen dynamischen Sprung an eine beliebige Programmposition, die zuvor in einer Variable gespeichert wurde. Dabei handelt es sich um eine spezielle Goto-Anweisung, die bereits in FORTRAN [5] vorhanden war

[1] http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Labels-as-Values.html

```
1  int count = 10;
2
3  active class Counter {
4      actions {
5          while (count > 0) {
6              count = count - 1;
7              yield;
8          }
9      }
10 }
11
12 void main() {
13     activate new Counter();
14     activate new Counter();
15     yield;
16 }
```

Listing 4.1: DBL-Beispielprogramm Counter

und dort als *Assigned Goto* bezeichnet wird. Im Gegensatz zu einem Goto aus C, dessen Sprungziel immer statisch zur Compile-Zeit festgelegt wird, erlaubt *Labels as Values* eine dynamische Programmfortführung in Abhängigkeit einer gespeicherten Label-Adresse.

Bei dem Konzept *Labels as Values* handelt es sich um eine nicht-standardisierte Erweiterung von C++. Es wird jedoch neben der *GNU Compiler Collection* (GCC) auch von anderen bekannten Compilern für C++ unterstützt. Dazu gehören die Compiler: IBM XL C/C++ für Linux v9.0, Clang v5.0[2] und Intel C++ Compiler (ICC) v14.

Das Konzept *Labels as Values* besteht aus folgenden Sprachelementen:

- der Goto-Anweisung,
- einer Label-Definition vor einer Anweisung,
- dem Operator && für den Erhalt der Adresse eines Label,
- dem Operator * für den Erhalt eines Label aus einer Adresse
- und einer Variable vom Typ void* für das Speichern und Lesen einer Label-Adresse.

Die Programmposition, an die ein Wechsel erfolgen soll, wird mit einem Label markiert. Das Label muss sich vor einer Anweisung an der betreffenden Programmposition befinden. Die Adresse des Labels wird zunächst in einer Variablen vom Typ void* gespeichert, wobei der Erhalt der Label-Adresse mit dem Operator && erfolgt. Der Wechsel der Ausführungsposition erfolgt mit der goto-Anweisung und der gespeicherten Label-Adresse, die dazu mit dem Operator * in das Label übersetzt wird.

Ein Beispiel für die Verwendung ist in Listing 4.2 dargestellt. Die Programmposition an der Anweisung printf wird hier mit dem Label l1 markiert. Die Adresse des

[2] Clang v5.0 basiert auf der Low Level Virtual Machine (LLVM) Infrastructure v3.3.

Labels &&l1 wird in der Variablen cont gespeichert. Danach wird die Adresse in der nachfolgenden Goto-Anweisung in das Label l1 übersetzt und der Sprung zum Label wird vollzogen. Das Programm gibt nur die Nachricht continued aus und wird dann beendet.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      void* cont;
6
7      cont = &&l1;
8      goto *cont;
9      cout << "skipped" << endl;
10     l1: cout << "continued" << endl;
11
12     return 0;
13 }
```

Listing 4.2: Ein Beispiel zur Verwendung von *Labels as Values* für einen Wechsel der Ausführungsposition.

4.1.2 Wechsel der Ausführungsposition mit der Switch-Anweisung

Eine andere Möglichkeit ist die Verwendung der Switch-Anweisung. Diese wird mit einer Variable aufgerufen und springt je nach Wert der Variable zu einem passenden Case-Zweig. Wenn kein Case-Zweig den Wert der Variable enthält, so wird der default-Zweig gewählt. Bei einer Switch-Anweisung können die möglichen Sprungziele also mit Werten an Case-Zweigen markiert werden. Die Switch-Anweisung hat den Vorteil, dass ihre Verwendung C++-standardkonform ist.

Ein Beispiel ist in Listing 4.3 dargestellt. Das Sprungziel wird in der Variablen cont hinterlegt. Die Switch-Anweisung wertet cont aus und springt zum Case-Zweig für den Wert 2. Es erfolgt die Ausgabe continued und das Programm wird beendet.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int cont;
6
7      cont = 2;
8      switch (cont) {
9          case 1:
10             cout << "skipped" << endl;
11         case 2:
12             cout << "continued" << endl;
13     }
14
15     return 0;
16 }
```

Listing 4.3: Ein Beispiel zur Verwendung einer Switch-Anweisung für einen Wechsel der Ausführungsposition.

4.1.3 Herausforderungen

Bei der Verwendung von *Labels as Values* und *Switch* gibt es zwei Probleme, deren Lösung in den nachfolgenden Abschnitten beschrieben wird.

Problem 1 Mit *Labels as Values* und *Switch* sind nur Sprünge innerhalb derselben Funktion erlaubt. Ein Kontextwechsel von einem Prozess zu einem anderen Prozess kann deshalb nur innerhalb einer C++-Funktion realisiert werden. Wie können aber die vielfältigen Prozessdefinitionen eines Simulationsmodells auf dieselbe C++-Funktion abgebildet werden, um Kontextwechsel mit *Labels as Values* und *Switch* durchzuführen?

Problem 2 Eine Prozessdefinition kann Funktionsaufrufe enthalten, die Scheduling-Anweisungen ausführen. Für Funktionen, die aus Prozessdefinitionen aufgerufen werden, müssen deshalb ebenfalls Kontextwechsel durchgeführt werden. Diese Funktionen können nicht auf C++-Funktionen abgebildet werden, sondern müssen auf die eine C++-Funktion abgebildet werden, die auch die Prozessdefinitionen enthält. Wie können Funktionen und Funktionsaufrufe in C++ emuliert werden, wenn eine Abbildung auf C++-Funktionen nicht erlaubt ist?

4.2 Abbildung von Prozessdefinitionen

Sämtliche Prozessverhaltensbeschreibung werden auf eine C++-Funktion mit dem Namen simulate abgebildet. In dieser Funktion wird für jede Verhaltensbeschreibung eine Markierung erzeugt, an der die Ausführung eines Prozesses beginnt. Je nachdem, ob ein Positionswechsel mit *Labels as Values* oder mit *Switch* erfolgt, wird eine Markierung mit einem Label oder mit einem Case-Zweig realisiert. Für jeden Aufruf einer Scheduling-Anweisung, die in einer Verhaltensbeschreibung verwendet wird, ist eine weitere Markierung erforderlich. Diese Markierung wird direkt nach der Scheduling-Anweisung gesetzt, so dass ein Prozess bei Erreichen der Scheduling-Anweisung an dieser Stelle später fortgesetzt werden kann.

Jede Scheduling-Anweisung wird auf mehrere C++-Anweisungen abgebildet. Zuerst wird die erzeugte Markierung im aktuellen Prozesskontext gespeichert. Der aktuelle Prozess wird außerdem in einer passenden Liste des Schedulers hinterlegt. Im Anschluss wird der nächste auszuführende Prozess mit Hilfe des Schedulers bestimmt und es erfolgt der Wechsel an die gespeicherte Ausführungsposition dieses Prozesses.

Der Wechsel erfolgt bei der Verwendung von *Labels as Values* direkt mit der Goto-Anweisung. Bei einem *Switch* erfolgt zunächst ein Sprung mit der Goto-Anweisung an ein festes Label, das sich direkt vor der Switch-Anweisung befindet. Danach erfolgt die Verzweigung mit der Switch-Anweisung.

Der Prozesskontext wird in einem Objekt einer Klasse Execution gespeichert. Dieses Objekt enthält die Ausführungsposition für eine spätere Fortsetzung des Prozesses. Die aktive Klasse, die Struktur und Verhalten eines Prozess beschreibt, wird auf eine C++-Klasse abgebildet, die von der Klasse Execution erbt. Somit kann für jeden Prozess eine Ausführungsposition gespeichert werden.

Die alternative Abbildung eines Wechsels der Ausführungsposition auf der Basis der Switch-Anweisung erfolgt analog. Dabei wird für jede Scheduling-Anweisung ein Case-Zweig innerhalb derselben Switch-Anweisung erzeugt. Je nach Simulationsmodell können auf diese Weise sehr viele Case-Zweige auftreten. Der C++-Standard[3] macht keine genauen Angaben zur Anzahl der erlaubten Case-Zweige. Empfohlen wird lediglich im Anhang B ein Minimum von 16384 Case-Zweigen. Je nach Compiler kann die Anzahl sogar unbeschränkt sein. Beim GCC[4] ist die Anzahl der Case-Zweige nur durch den Speicher selbst beschränkt. Selbst ein Modell mit 16384 Scheduling-Anweisungen wäre sehr umfangreich und dürfte nur schwer zu erreichen sein. Der Einsatz der Switch-Anweisung stellt demnach eine Alternative zur Verwendung von *Labels as Values* dar.

4.2.1 Beispiel mit *Labels as Values*

Listing 4.4 zeigt ein Beispiel für die Abbildung von Prozessverhaltensbeschreibungen. Dabei wird das DBL-Modell aus Listing 4.1 nach C++ unter Verwendung von *Labels as Values* abgebildet. Die Klasse Counter wird auf eine C++-Klasse abgebildet, die von der Klasse GotoExecution erbt. Die Klasse GotoExecution ist in diesem Fall auf die Verwendung mit *Labels as Values* zugeschnitten. Sie stellt einen Konstruktor bereit, dem die Adresse eines Labels übergeben werden kann, an der die Ausführung eines Prozesses beginnt. Die Implementierung der im Folgenden verwendeten C++-Klassen befindet sich vollständig im Anhang dieser Arbeit A.2.

Die Verhaltensbeschreibung der main-Funktion und der Klasse Counter befinden sich in der Funktion simulate, mit der die Ausführung des C++-Programms beginnt. Die simulate-Funktion beginnt mit der Erzeugung des Schedulers sowie einer Prozessausführung für die main-Funktion des DBL-Modells. Die globale Variable cx dient dabei der Speicherung des aktuell aktiven Prozesses. Das GotoExecution-Objekt der main-Funktion wird der Liste ml (moving list) hinzugefügt. Des Weiteren werden die GotoExecution-Objekte der beiden Counter-Prozesse der Liste ml hinzugefügt.

Die yield-Anweisung der main-Funktion wird auf die Anweisungen in den Zeilen 13 bis 16 abgebildet. Der Aufruf der Funktion yield() am Scheduler gibt dabei den nächsten Prozess in der Liste ml zurück. Dieser wird zum aktiven Prozess und die Ausführung wechselt mit der Goto-Anweisung zur gespeicherten Programmposition. Die main-Funktion endet nach der yield-Anweisung mit einem Sprung an das Ende der simulate-Funktion.

Zunächst wechselt die Ausführung jedoch zum ersten der Counter-Objekte. Die Verhaltensbeschreibung der Klasse Counter beginnt am Label Counter_behavior. Es folgt die While-Schleife, die die count-Variable prüft und ihren Wert um eins verringert. Im Anschluss erfolgt die Abbildung für die Yield-Anweisung. Die Ausführungsposition für die Fortsetzung wird mit Hilfe des Labels Counter_behavior_l1 markiert und im aktuellen Kontext gespeichert. Am Ende der Verhaltensbeschreibung erfolgt der Aufruf der Funktion terminate() am Scheduler. Dieser Aufruf gibt den vom GotoExecution-Objekt belegten Speicher frei. Die Ausführung wird danach mit dem nächsten wartenden Prozess fortgesetzt.

[3] https://isocpp.org/std/the-standard
[4] http://gcc.gnu.org/onlinedocs/gcc/Statements-implementation.html

```
1  GotoExecution* cx;
2  int count = 3;
3
4  void simulate() {
5      Scheduler* sched = new Scheduler();
6      cx = new GotoExecution(0, &&main_l1);
7      sched->add(cx);
8      main_l1:;
9
10     sched->add(new Counter(1, &&Counter_behavior));
11     sched->add(new Counter(1, &&Counter_behavior));
12
13     cx->cont = &&main_l2;
14     cx = sched->yield();
15     goto *(cx->cont);
16     main_l2:;
17
18     goto end_of_main;
19
20     Counter_behavior:;
21     while (count > 0) {
22         count--;
23         cx->cont = &&Counter_behavior_l1;
24         cx = sched->yield();
25         goto *(cx->cont);
26         Counter_behavior_l1:;
27     }
28     cx = sched->terminate();
29     goto *(cx->cont);
30
31     end_of_main:;
32 }
```

Listing 4.4: Beispiel für die Abbildung eines DBL-Modells nach C++ mit GotoSwitching und *Labels as Values*.

4.3 Abbildung von Funktionsaufrufen auf einen emulierten Call-Stack

Eine Prozessverhaltensbeschreibung wird im Allgemeinen auf mehrere Funktionen aufgeteilt, die bestimmte Aspekte des Verhaltens kapseln. Verwendet eine solche Funktion selbst Scheduling-Anweisungen, so kann sie nicht auf eine C++-Funktion abgebildet werden. Dies hätte mit dem beschriebenen Ansatz für den Wechsel der Ausführungsposition einen Sprung von einer Funktion zu einer anderen Funktion zur Folge. Ein solcher Sprung resultiert in nicht definiertem Verhalten und muss vermieden werden. Funktionen, die Scheduling-Anweisungen enthalten oder andere Funktionen aufrufen, die Scheduling-Anweisungen enthalten, werden deshalb auf C++-Anweisungen abgebildet, die einen Call-Stack nachbilden. Diese Funktionen können durch eine statische Analyse der Funktionsaufrufe ermittelt werden. Alle anderen Funktionen können direkt auf C++-Funktionen abgebildet werden.

Für die Emulation wird der Call-Stack als Teil des Prozesskontextes gespeichert. Die

```
1  class Execution {
2      ...
3      char mem[STACK_SIZE];
4      char* top;
5
6      union lrv_union {
7          int iv; double dv; bool bv; void* pv;
8      } lrv;
9
10     void push(void* returnPoint, int variablesSize) {
11         ((class Frame*)(top))->returnPoint = returnPoint;
12         top = top + sizeof(class Frame) + variablesSize;
13     }
14
15     void pop(int vsize) {
16         top = top - sizeof(class Frame) - variablesSize;
17     }
```

Listing 4.5: Die für die Stack-Emulation relevanten Teile der Klasse Execution.

dafür relevanten Variablen und Funktionen werden in der Klasse Execution definiert, die in Listing 4.5 dargestellt wird. Die Definition des Call-Stacks besteht aus

1. einer Variable mem, die einen Speicherbereich für Stack-Frames reserviert,
2. einer Variable top, die auf den freien Speicher am Ende des Stacks verweist,
3. einer Variable lrv, die für den Rückgabewert einer Funktion verwendet wird und
4. den Funktionen push und pop mit denen Stack-Frames erzeugt und entfernt werden.

Die Funktion push erlaubt eine effiziente Erzeugung neuer Stack-Frames, da ein passender Speicherbereich nicht erst gesucht werden muss, sondern direkt verfügbar ist. Die Größe des Stacks ist dabei jedoch fest und kann während der Ausführung nicht verändert werden.

4.3.1 Abbildung

Jede Funktion wird wie folgt analog zu den bereits beschriebenen Prozessverhaltensbeschreibungen abgebildet. Eine Funktion erhält eine Marke an der ihre Ausführung beginnt. Der Aufruf der Funktion erfolgt mit einem Sprung an diese Marke. Nach der Anweisung für den Sprung befindet sich eine weitere Marke, an die ein Sprung erfolgt, sobald die aufgerufene Funktion zurückkehrt. Die Adresse dieser Marke wird in einem Stack-Frame-Objekt gespeichert, das durch Aufruf der Funktion push erzeugt wird. Dabei wird das aktuelle Ende des Stacks als Stack-Frame-Objekt interpretiert, ohne das ein Objekt erzeugt werden muss. Die Stack-Frame-Klasse wird in Listing 4.6 gezeigt.

Die lokalen Variablen und Parameter einer Funktion werden ebenfalls in dem Speicherbereich für den erzeugten Stack-Frame abgelegt. Dazu erhält die Funktion push

```
class Frame {
    public: void* returnPoint;
};
```

Listing 4.6: Definition der Klasse für Stack-Frames.

eine Angabe über die Größe des Speicherbereichs für die Variablen der zugrundeliegenden Funktion und setzt das Ende des Stacks auf den freien Speicher für den nächsten Stack-Frame. Ausgehend vom neuen Ende des Stacks in der Variablen top werden die Zugriffe auf die Variablen der Funktion durch Zeiger-Arithmetik berechnet. Die so berechneten Adressen der Variablen werden dann entsprechend ihres Typs passend interpretiert, um so auf Werte korrekt zuzugreifen. Die Variablen belegen den Speicher im Stack-Frame in der Reihenfolge ihrer Definition in der zugrundeliegenden Funktion. Der Rückgabewert einer Funktion wird separat in der Variablen lrv gespeichert und passend interpretiert. Das Speicher-Layout für den Stack ist in Abbildung 4.1 dargestellt.

4.3.2 Beispiel

Listing 4.7 zeigt eine Beispiel-Funktion m in DBL. Die Funktion hat einen Parameter d vom Typ int und eine Variable a vom Typ bool. Die Variable a befindet sich im Stack-Frame nach der Variablen d. Der Zugriff auf die Variable a erfolgt durch einen komplexen Ausdruck, der in Listing 4.8 abgebildet ist. Das Ende des Stacks in top ist vom Typ char*. Damit ist es möglich, durch Zeiger-Arithmetik im Abstand der Größe von einem Byte im Speicher zu navigieren. Zunächst wird die Größe der Variablen der Funktion m von der Adresse top abgezogen. Danach wird die Größe der Variablen d addiert, um zu der Position der Variablen a zu gelangen. Am Ende wird diese Adresse als ein Zeiger auf ein bool interpretiert.

```
void m(int d) {
    bool a = false;
    if (d > 0) {
        m(d-1);
    } else {
        yield;
        a = true;
    }
}
```

Listing 4.7: Eine Beispiel-Funktion m in DBL.

```
(* reinterpret_cast<bool*>(
    cx->top - (sizeof(int) + sizeof(bool)) + sizeof(int)
))
```

Listing 4.8: Der Ausdruck für den Zugriff auf eine lokale Variable vom Typ bool als Teil eines emulierten Stacks.

Ein Zugriff auf eine Variable ist mit dem emulierten Stack nun wesentlich komplizierter. Dies ist jedoch kein Nachteil, da der Ausdruck das Ergebnis einer automatischen Abbildung ist.

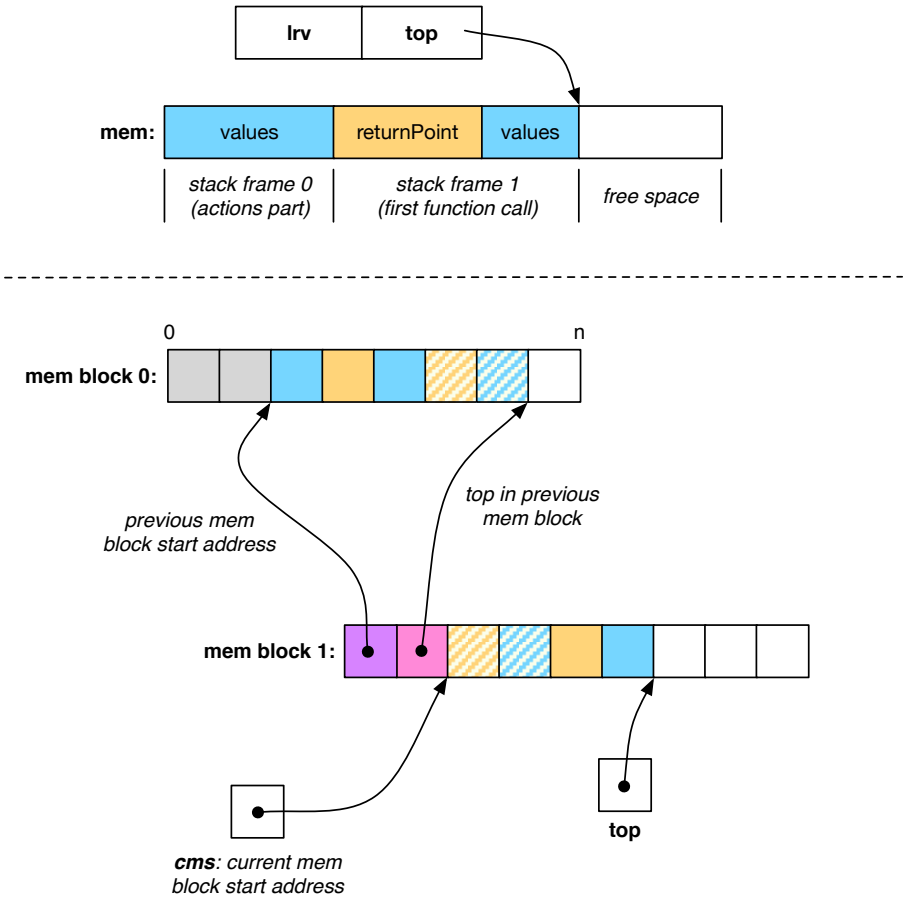

Abbildung 4.1: Das Speicher-Layout des Call-Stacks, der von links nach rechts wächst.

```
1  void simulate() {
2      ...
3      fn_m:;
4      M_A = false;
5      if (M_D > 0) {
6          M_D_next = M_D - 1; // setzt das Argument d für den Aufruf von m
7          cx->push(&&fn_m_l2, M_values_size);
8          goto fn_m;
9          fn_m_l2:;
10     }
11     else {
12         cx->cont = &&fn_m_l1;
13         cx = sched->yield();
14         goto *(cx->cont);
15         fn_m_l1:;
16         M_A = true;
17     }
18     cx->pop(M_values_size);
19     goto *( ((class Frame*)(cx->top))->returnPoint );
20     ...
```

Listing 4.9: Emulation der Aufrufe der Beispiel-Funktion m.

Das vollständige Ergebnis für die Abbildung der Funktion m mit dem emulierten Stack ist in Listing 4.9 zu sehen. Die Ausdrücke für den Zugriff auf die Variablen sind mit den C-Makros M_D, M_D_next und M_A vereinfacht. Diese Makros definieren Zugriffe entsprechend des Zugriffs auf die Variable a im Listing 4.8. Dabei erfolgt mit M_D der Zugriff auf die Variable d im aktuellen Funktionsaufruf. Mit dem Makro M_D_next erfolgt hingegen der Zugriff auf die Variable d kurz vor dem Zeitpunkt der Erzeugung eines neuen Stack-Frames für die Funktion m. Auf diese Weise können die Werte von Funktionsargumenten hinterlegt werden. Die Größe des Speichers für die Variablen wird durch das Makro M_values_size definiert.

4.4 Verwandte Implementierungen für Prozesskontextwechsel

4.4.1 SLX

Die schnellste bekannte Durchführung von Prozessenkontextwechseln erfolgt durch den Programmcode, der vom Compiler der Sprache SLX erzeugt wird. Zur Implementierung des SLX-Compilers sind jedoch kaum Informationen vorhanden, da der Compiler nicht offen zugänglich ist. Bekannt ist, dass der Compiler für SLX ein Programm in eine nicht näher spezifizierte Assembler-Sprache von Microsoft übersetzt. SLX-Programme sind somit nur unter dem Betriebssystem Windows lauffähig, was bereits eine starke Einschränkung auf eine spezifische Plattform für die Ausführung von Simulationen bedeutet.

Der SLX-Compiler ist für besonders schnelle Kontextwechsel zwischen Prozessen optimiert. Er verwendet jedoch keine Techniken zur automatischen Optimierung beliebigen Programmcodes, wie sie von C++-Compilern eingesetzt werden. Ein C++-

Compiler sollte demnach einen Programmcode erzeugen, der im Allgemeinen schneller als der vom SLX-Compiler erzeugte Programmcode ist. In Bezug auf Kontextwechsel zwischen Prozessen, die im Umfeld von C++ keine bedeutende Rolle spielen, sollte der von einem C++-Compiler erzeugte Programmcode im Allgemeinen jedoch langsamer als der Programmcode des SLX-Compilers sein.

4.4.2 C++

Inline-Assembler-Code

Für die Implementierung von Kontextwechseln in C++ gibt es eine Reihe von Alternativen. Die älteste bekannte Möglichkeit ist die Verwendung von Inline-Assembler-Code, der Registerinhalte und Speicher direkt verändert, um so einen Kontextwechsel zu realisieren. Diese Möglichkeit wurde in der C-Erweiterung *C with Classes* [73] [74], die später als C++ bekannt wurde, von Stroustrup implementiert. Eine Verwendung von Inline-Assembler-Code besitzt den Nachteil, dass der Code abhängig von einer konkreten Prozessorarchitektur ist. Eine solche Abhängigkeit sollte vermieden werden, da sie mit hohen Kosten für Wartung und Portierung verbunden ist, die für Simulationssprachen mit ihrem eingeschränkten Entwicklerkreis hohe Kosten darstellen.

setjmp/longjmp

Die erste plattformunabhängige Implementierung von Kontextwechseln für einen Simulationskern in C++ wurde von Hansen [31] beschrieben. Er verwendet die Funktionen setjmp und longjmp, die zur C-Standardbibliothek gehören. Die Funktion setjmp speichert den aktuellen Zustand der Ausführung in einer Datenstruktur vom Typ jmp_buf. Der Zustand umfasst den Programmzähler, die Adresse des aktuellen Endes des Stacks und die Werte relevanter CPU-Register. Mit setjmp wird also ein Ausführungszustand gespeichert zu dem ein Programm später zurückkehren kann. Dieser Zustand wird beim Aufruf der Funktion longjmp wiederhergestellt. Dabei wird die Adresse für das Ende des Stacks zurückgesetzt. Alle in der Zwischenzeit erzeugten Stack-Frames für aufgerufene Funktionen werden nun überschrieben.

Die Funktionen setjmp und longjmp besitzen eine Einschränkung. Sie operieren auf einem Stack im Speicher, der sich an einer bestimmten nicht veränderbaren Stelle befindet. Es ist nicht möglich einen weiteren Stack an einer anderen Stelle im Speicher zu erzeugen und zu verwenden. Eine prozessorientierte Simulation besteht jedoch aus vielen Prozessen, die jeweils ihren eigenen Stack benötigen. Deshalb ist für die Sicherung des aktuellen Stacks das Kopieren an eine andere Stelle im Speicher notwendig. Für die Wiederherstellung muss ein zuvor gespeicherter Stack an die Stelle des aktuellen Stacks kopiert werden. Das Kopieren von Speicher erhöht die Zeit für die Ausführung von Kontextwechseln. Beispiele für Simulationsbibliotheken, die Kontextwechsel auf der Basis von setjmp und longjmp einsetzen sind ODEMx [28] und COROUTINE [32].

Für die Bibliothek COROUTINE gibt es eine alternative Implementierung, bei dem sich mehrere Prozesse einen Stack teilen. Dabei wird der Stack in mehrere Teile zerlegt. Jeder Prozess verwendet einen bestimmten Teil des Stacks. Die Größe des Stacks ist bei dieser Variante unveränderlich.

Fibers

Fibers sind leichtgewichtige User-Level-Threads, die von einem Betriebssystem über eine spezifische Schnittstelle bereitgestellt werden. Unter dem Betriebssystem Windows sind Fibers als Windows Fibers bekannt. Dagegen sind Fibers unter POSIX-konformen Betriebssystemen wie z.B. Unix und Linux als setcontext/getcontext oder ucontext bekannt. Im Unterschied zu setjmp/longjmp wird bei Fibers für jede Ausführung ein separater Stack erzeugt und verwendet. Damit ist das Kopieren des Stacks hier nicht notwendig. Ein Beispiel für eine Simulationsbibliothek, die Fibers verwendet ist ODEMx [28].

Protothreads

Protothreads [21] sind ebenfalls leichtgewichtige User-Level-Threads, die für kooperatives Multitasking in speicherbeschränkten eingebetteten Systemen eingesetzt werden. Protothreads teilen sich per Definition einen gemeinsamen Call-Stack. Es wird nicht versucht, den Call-Stack bei einem Kontextwechsel zu sichern und wiederherzustellen. Stattdessen verwenden Protothreads globale Variablen, um sich ihren Zustand zu merken. Protothreads stellen somit keine vollständige Implementierung von Prozesskontextwechseln dar. Ich erwähne sie dennoch, da hier zum ersten Mal das C++-Konzept *Labels as Values* für den Wechsel der Ausführungsposition eingesetzt wird und ich dieses auch als Grundlage für den von mir entwickelten Ansatz einsetze.

4.4.3 Java

Neben Laufzeitvergleichen von Kontextwechseln in SLX und in C++ wird auch Java als stark verbreitete Programmiersprache betrachtet. In Java werden Kontextwechsel häufig mit Threads implementiert, die in der Ausführung sehr langsam sind. Mit JiST [7] gibt es jedoch eine Java-basierte Simulationsbibliothek, die auf einem Byte-Code-Rewriting basiert und eine schnellere Ausführung erlaubt.

JiST ist besonders für Netzwerksimulationen verbreitet und in diesem Bereich sogar schneller als C++-basierte Simulatoren, wie ein Vergleich belegt [83] [65]. Die Implementierung von Kontextwechseln erfolgt in JiST durch ein Umschreiben des Java-Byte-Code. Dabei wird die Bytecode-Instruktion goto eingesetzt und der Call-Stack für Funktionsaufrufe emuliert. Spezielle Stack-Frame-Objekte werden auf dem Heap erzeugt. Wenn ein Prozess fortgesetzt werden soll, wird sein Call-Stack neu aufgebaut. Dazu werden alle zuvor aufgerufenen Funktionen erneut betreten und die Werte der Variablen auf dem Stack werden wiederhergestellt.

JiST unterstützt eine prozessorientierte Modellierung nicht vollständig. Es fehlen Scheduling-Anweisungen für das Warten auf eine Reaktivierung durch einen anderen Prozess und für die Weitergabe der Steuerung an einen anderen Prozess. Für die Durchführung der Laufzeitvergleiche wurde deshalb mit JiST-Pro[5] eine JiST-Erweiterung implementiert, die die fehlenden Scheduling-Anweisungen enthält.

[5] http://github.com/ablunk/dmx/tree/master/dev-plugins/JiST-Pro

4.5 Bewertung

Die Laufzeiteffizienz von Prozesskontextwechseln mit GotoSwitching in den Varianten *Labels as Values* und *Switch* wird durch Laufzeitvergleiche mit Implementierungen für Prozesskontextwechseln in C++, Java und SLX bewertet. Da die Methode emulierte Funktionen verwendet, wird zusätzlich die Laufzeit von emulierten Funktionsaufrufen untersucht. Als dritte Variante wird eine Kombination betrachtet, bei der Kontextwechsel in einer Funktion mit einem großen Call-Stack durchgeführt.

In den Laufzeitvergleichen werden folgenden Implementierungen für Prozesskontextwechsel untersucht:

- COROUTINE mit setjmp/longjmp in der Variante kopierter Stack,
- COROUTINE mit setjmp/longjmp in der Variante geteilter Stack,
- ODEMx mit Windows Fibers,
- Minimalkern mit Windows Fibers (Min/Fibers),
- GotoSwitching mit *Labels as Values* (GotoSwitching/LaV),
- GotoSwitching mit *Switch* (GotoSwitching/Switch),
- SLX
- und JiST.

Jede Variante basiert auf einem DBL-Referenzmodell, das auf jede untersuchte Implementierung übertragen wurde. Die verwendeten Modelle sind online verfügbar[6]. Für eine bessere Vergleichbarkeit zwischen GotoSwitching und Windows Fibers wird neben ODEMx ein minimaler Simulationskern mit Windows Fibers eingesetzt (Min/-Fibers), der nur Kontextwechsel durchführen kann. Außerdem verwenden Min/Fibers und GotoSwitching den gleichen Scheduler, der Prozesse ohne Betrachtung einer Modellzeit in einer einzigen Prozessliste *Moving List* (ml) verwaltet (Scheduler-Implementierung siehe Anhang in Listing A.5).

4.5.1 Computersystem, Programmversionen und Messmethode

Die Messungen wurden auf einem MacBook Pro mit 2.6 GHz Intel Core i7 CPU und 8 GB Hauptspeicher durchgeführt. Dabei wurde nicht Mac OS sondern Microsoft Windows 8.1 64-Bit als Betriebssystem eingesetzt, um Ergebnisbeeinflussungen durch das Betriebssystem zu vermeiden.

Die C++-basierten Simulationsprogramme wurden in MinGW 32-Bit mit GCC G++ v4.8.1 ausgeführt. SLX wurde in version 2.3 OV283 verwendet. Die Java-basierte Simulationsbibliothek JiST-Pro wurde mit Oracle JDK 1.7.0 ausgeführt.

Die Ausführungszeit wurde für C++ mit der C-Funktion `gettimeofday`, für Java mit der Funktion `System.nanoTime()` und für SLX mit der Funktion `real_time()` gemessen. Die Messergebnisse werden mit Erwartungswert und Standardabweichung bei 20 Wiederholungen dargestellt.

[6] http://github.com/ablunk/dmx/blob/master/dbl-core/benchmarks.zip

4.5.2 Laufzeitmessungen

Kontextwechsel

Der erste Laufzeitvergleich untersucht ausschließlich Kontextwechsel. Dazu wird ein Modell mit zwei Prozessen verwendet, die wechselseitig sehr oft den Kontext wechseln. Das DBL-Referenzmodell entspricht dem bereits gezeigten Counter-Modell in Listing 4.1.

Bei einer Ausführung mit 10^8 Kontextwechseln ergibt sich eine Ausführungszeit, die für die schnellsten Implementierungen nicht zu kurz und für die langsamsten Implementierungen nicht zu lang ist. Das Ergebnis mit den Ausführungszeit wird in Tabelle 4.1 dargestellt.

GotoSwitching mit Switch hat die kürzeste Ausführungszeit und wird dicht gefolgt von SLX. Obwohl die Abbildung nach C++ und nicht nach Assembler erfolgt, ist die Ausführungszeit schneller als die von SLX. In der Variante *Labels as Values* ist GotoSwitching nur 4% langsamer als SLX.

Der Minimalkern mit Windows Fibers ist dagegen bereits 48% langsamer als GotoSwitching mit *Labels as Values*. Die setjmp-basierte Implementierung COROUTINE ist langsamer als der Minimalkern mit Windows Fibers. Der Abstand zum Fibers-basierten ODEMx ist groß. ODEMx ist 18 mal langsamer als der Minimalkern mit Windows Fibers und sogar langsamer als das Java-basierte JiST-Pro.

Tabelle 4.1: Erwartungswert μ und Standardabweichung σ der Ausführungszeit von 10^8 Kontextwechseln (in Sekunden) und Kontextwechselrate (in eine Million Kontextwechsel pro Sekunde [10^6 W/s]).

Implementierung	**Zeit [s]**		**Wechselrate**
	μ	σ	**[10^6 W/s]**
ODEMx (Fibers)	29.24	0.23703	3.42
JiST-Pro	14.33	0.60959	6.98
COROUTINE (kopierter Stack)	11.08	0.02487	9.03
COROUTINE (geteilter Stack)	4.26	0.00021	23.50
Min/Fibers	1.62	0.00004	61.57
GotoSwitching/LaV	0.85	0.00008	117.53
SLX	0.82	0.00006	122.44
GotoSwitching/Switch	0.81	0.00003	122.72

Funktionsaufrufe

Das zweite Referenzmodell, dargestellt in Listing 4.10, dient der Ermittlung der Ausführungszeit von Aufrufen emulierter Funktionen im Vergleich zu nativen C++-Funktionen und zu Funktionen in SLX. ODEMx und JiST-Pro werden nicht weiter betrachtet, da ihre Ausführungszeit bei Kontextwechseln bereits sehr langsam ist.

Die C++-Implementierung wird mit den Compiler-Flags `-fno-inline` and `-O3` übersetzt. Das erste Flag sorgt dafür, dass die Funktion auch tatsächlich aufgerufen wird.

```
1  int totalCalls = 800000000;
2
3  int f(int n) {
4      return n+1;
5  }
6
7  void main() {
8      int sum = 0;
9      for (int i = 0; i < totalCalls; i++) {
10         sum += f(i);
11     }
12 }
```

Listing 4.10: Referenzmodell für Funktionsaufrufe in DBL ohne Messanweisungen.

Das zweite Flag aktiviert Optimierungen auf Stufe 3. Für GotoSwitchting werden die emulierten Funktionen nur mit dem Flag -O3 übersetzt.

Die Ausführungszeit wird für $8 * 10^8$ Funktionsaufrufe bestimmt. Das Ergebnis wird in Tabelle 4.2 dargestellt. Die emulierten Funktionen werden am schnellsten ausgeführt und sind etwa 4 mal schneller als SLX-Funktionen. Sie sind aber auch schneller als native C++-Funktionen.

GotoSwitchting verwendet einen feste Größe für den Call-Stack. Dies kann in Simulationen mit einer hohen Anzahl von Prozessen zu einem Problem bezüglich des Gesamtspeichers durch eine zu geringe Speicherausnutzung werden. Der Einbau einer Prüfung für das Erreichen des Stack-Endes fehlt und würde die Laufzeit negativ beeinflussen. Eine Fibers-basierte Implementierung hat hier Vorteile, da der Stack dynamisch in der Größe einer Speicherseite (page) wachsen kann und einen Zugriff über das Ende des Stacks hinaus mit einem Seitenfehler (page fault) erkannt wird.

Tabelle 4.2: Ausführungszeit von $8 * 10^8$ Funktionsaufrufen (in Sekunden) und Aufrufrate (in Millionen Aufrufe pro Sekunde [MF/s]).

Implementierung	**Zeit [s]**		**Aufrufrate**
	μ	σ	**[MF/s]**
SLX	5.28	0.01	151.43
C++ nativ, -fno-inline -O3	1.58	0.01	505.66
GotoSwitchting/LaV, -O3	1.35	0.01	590.51

Kombination

Der dritte Laufzeitvergleich bestimmt die Ausführungszeit von Kontextwechseln mit einem großen Call-Stack. Dazu wird das erste Referenzmodell angepasst. Die Anweisung yield wird durch einen Aufruf einer Funktion ersetzt, die ein Feld mit einer Größe von 128 KB erzeugt und danach erst einen Kontextwechsel ausführt.

Die C++-Implementierungen werden mit dem Compiler-Flag -O0 übersetzt, das Optimierungen deaktiviert. So wird sichergestellt, dass das Feld auch tatsächlich angelegt wird. Des Weiteren ist für die Vergleichbarkeit mit SLX eine Initialisierung des

angelegten Feldes mit Standardwerten erforderlich, da Variablen in SLX immer mit einem Standardwert initialisiert werden.

Die Laufzeit wird für $8 * 10^5$ Kontextwechsel bestimmt und in Tabelle 4.3 dargestellt. Wie man erwarten kann, ist die Implementierung in COROUTINE mit einem kopierten Stack am langsamsten. Die Laufzeitunterschiede der anderen Implementierungen sind dagegen nur gering, da der Call-Stack hier nicht kopiert wird. Die Anzahl der Kontextwechsel ist hier zu gering, um eine Aussage zu Laufzeitunterschieden zu treffen. Diese wurden bereits mit dem ersten Laufzeitvergleich (siehe 4.5.2) gezeigt.

Tabelle 4.3: Ausführungszeit von $8 * 10^5$ Kontextwechseln mit einem Stack der Größe 128 KB (in Sekunden) und der Kontextwechselrate (in eintausend Kontextwechsel pro Sekunde [1000 W/s]).

Implementierung	**Zeit [s]**		**Wechselrate**
	μ	σ	**[1000 W/s]**
COROUTINE (kopierter Stack)	12.70	0.06	63.01
SLX	3.21	0.01	249.27
Min/Fibers	3.11	0.01	257.21
COROUTINE (geteilter Stack)	3.09	0.01	259.23
GotoSwitching/LaV	2.99	0.01	267.30

5 Spracherweiterungsansatz DMX

Der Ansatz *Discrete-Event Modelling with Extensibility* (DMX) ist ein Spracherweiterungsansatz für metamodellbasierte Simulationsbasissprachen, der in diesem Kapitel allgemein für beliebige Basissprachen vorgestellt wird. Der Ansatz führt die Syntaxdefinition eines Domänenkonzeptes auf eine Erweiterung der Syntaxdefinition der Basissprache zurück. Die Semantik eines Domänenkonzeptes ist eine Reduktion auf Basiskonzepte, die durch eine Abbildung in der Basissprache auf die Basissprache definiert wird.

Ich gebe zunächst einen Überblick zu den Komponenten des Ansatzes. Danach beschreibe ich das in DMX verwendete Sprachkonzept zur Definition von Erweiterungen. Die Realisierung von Erweiterungen erfolgt mit einer Syntaxerweiterung und einer Konzeptreduktion, die im Anschluss erklärt werden. Abschließend vergleiche ich DMX mit ähnlichen Ansätzen in Bezug auf die effiziente Entwicklung einer DSL.

5.1 Überblick

DMX besteht aus vier allgemeinen Komponenten, die unabhängig von konkreten Implementierungstechnologien sind. Diese allgemeinen Komponenten werden durch das Framework DMX implementiert. Abb. 5.1 stellt die Zusammenhänge zwischen den allgemeinen Komponenten und ihrer jeweiligen Implementierung dar. Ich beschreibe zunächst die allgemeinen Komponenten und danach die Implementierung.

Alle Erklärungen in diesem Kapitel erfolgen unter der Annahme, dass die abstrakte Syntax der Basissprache durch ein objektorientiertes Metamodell und die konkrete Syntax durch eine kontextfreie Grammatik definiert sind (siehe Kap. 2.4).

5.1.1 Allgemeine Komponenten

Die Erweiterung einer metamodellbasierten Basissprache lässt sich mit vier allgemeinen Komponenten realisieren:

1. einer objektorientierten *Basissprache*,
2. einer *Sprache für die Erweiterungsdefinition* als Teilsprache der Basissprache,
3. einem Algorithmus zur *Syntaxerweiterung*
4. und einem Algorithmus zur *Konzeptreduktion*.

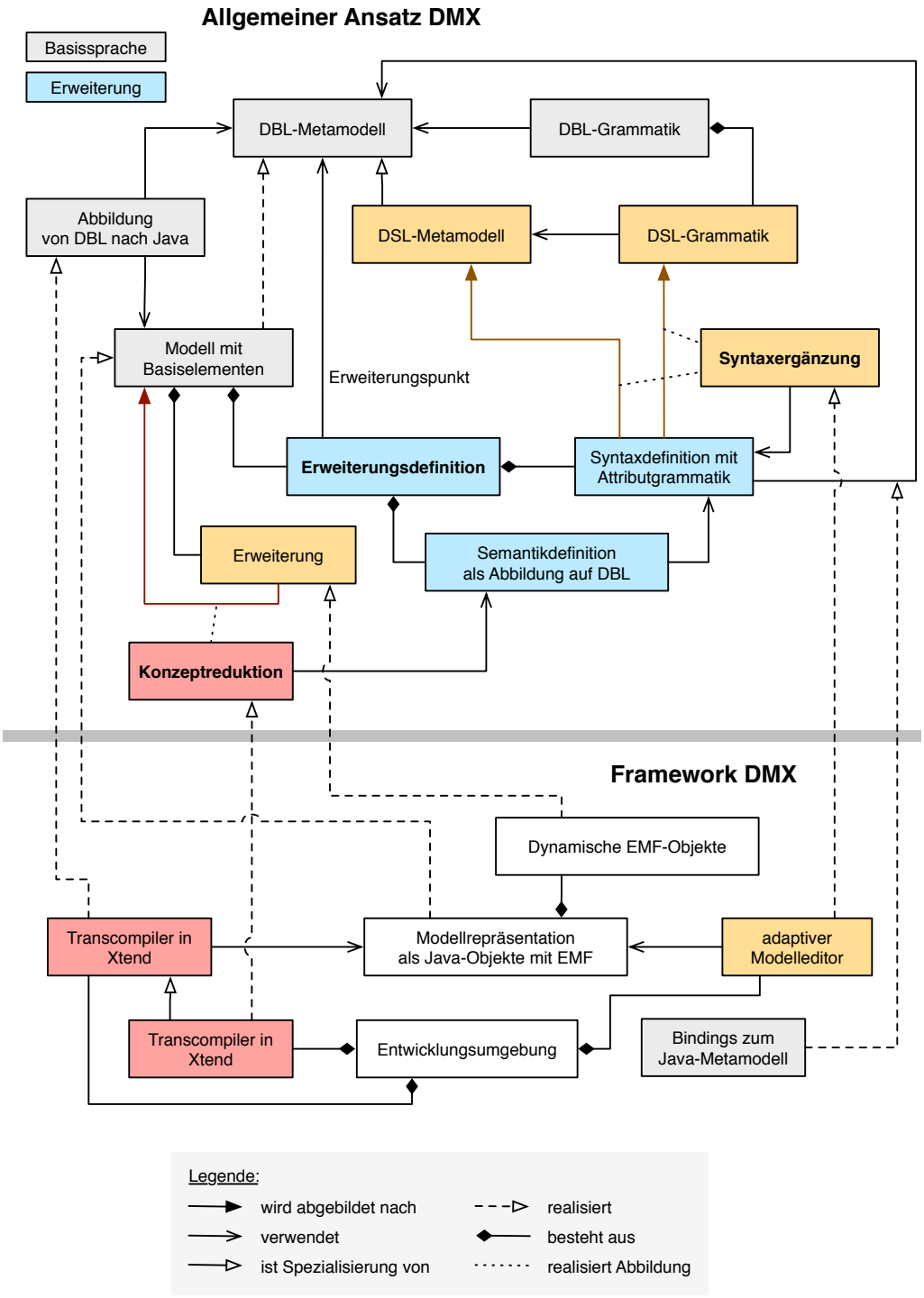

Abbildung 5.1: Allgemeine Komponenten des Ansatzes DMX und ihre Implementierung als Framework DMX.

Erweiterbare Basissprache

Als Basissprache kann eine beliebige objektorientierte Sprache verwendet werden, sofern diese mit einem Metamodell und einer Grammatik definiert ist. Die objektorientierte Basissprache bezeichne ich als *Object-oriented Base Language* (OBL). Die Basissprache enthält Basiskonzepte, die als Ausgangspunkt für die Modellierung dienen und durch Domänenkonzepte erweitert werden können.

Eine Erweiterung ist dann sinnvoll, wenn Modelle ein wiederkehrendes Muster enthalten, das sich mit einer passenden Syntax prägnanter als mit der Syntax der Basissprache darstellen lässt. Ein Beispiel für ein solches Muster ist die zustandsorientierte Modellierung eines reaktiven Systems mit den Konzepten der objektorientierten Programmiersprache C++ [59], an Stelle der Verwendung einer DSL für Zustandsautomaten.

Die Definition einer Erweiterung setzt geeignete Sprachkonzepte voraus, die zur Basissprache hinzugefügt werden. Da diese Sprachkonzepte zur Definition von Sprachkonzepten eingesetzt werden, bezeichne ich sie als Metakonzepte. Sie befinden sich auf dem gleichen Level wie die Sprachkonzepte von Metamodell- und Grammatiksprachen.

Definition 5.1 (Erweiterungsdefinition)**.** Eine Erweiterungsdefinition führt eine neue syntaktische Ausprägung für ein bestimmtes Basiskonzept ein und definiert dazu eine Syntax und eine Semantik.

Definition 5.2 (Erweiterungsinstanz)**.** Eine Erweiterungsinstanz ist eine Ausprägung einer Erweiterungsdefinition, entsprechend den Regeln der Syntaxdefinition. Die Semantik einer Erweiterungsinstanz ist ein Teilprogramm der Basissprache, das die Erweiterungsinstanz ersetzt.

Zur besseren Unterscheidung bezeichne ich die OBL-Teilsprache für die Erweiterungsdefinition als *Extension Definition Language* (EDL). Die Sprache EDL enthält ein Metakonzept zur Erweiterungsdefinition. Die Syntaxdefinition erfolgt dabei mit Metakonzepten, die ich zu einer weiteren Teilsprache mit der Bezeichnung *Extension Syntax Language* (ESL) zusammenfasse. Für die Semantikdefinition wird die Teilsprache *Extension Reduction Language* (ERL) verwendet, die die Konzepte von OBL und zusätzlich ein Konzept für die Definition der Abbildung von Erweiterungen auf OBL enthält. Der Zusammenhang zwischen OBL und den Teilsprachen wird in Abb. 5.2 verdeutlicht.

Eine objektorientierte Basissprache vereinfacht die Definition von Erweiterungen. Um Zugriffe auf ein Basiskonzept und seine Struktur in der Basissprache zu ermöglichen, wird die Metaklasse für ein Basiskonzept durch ein OBL-Interface in der Basissprache selbst repräsentiert. Damit lassen sich Verweise auf Basiskonzepte in einem in sich geschlossenen Modell realisieren, das nur Instanzen von Basiskonzepten enthält. Das Interface dient als Schnittstelle zur Metaklasse eines Basiskonzeptes und erlaubt i) die Angabe eines Basiskonzeptes als Teil einer Erweiterungsdefinition, ii) die Angabe von Basiskonzepten in einer Syntaxdefinition und iii) den Zugriff auf die Struktur eines Basiskonzeptes in einer Semantikdefinition. Ein Beispiel wird in Listing 5.1. Hier wird die Metaklasse für das Konzept Funktion als Interface dargestellt.

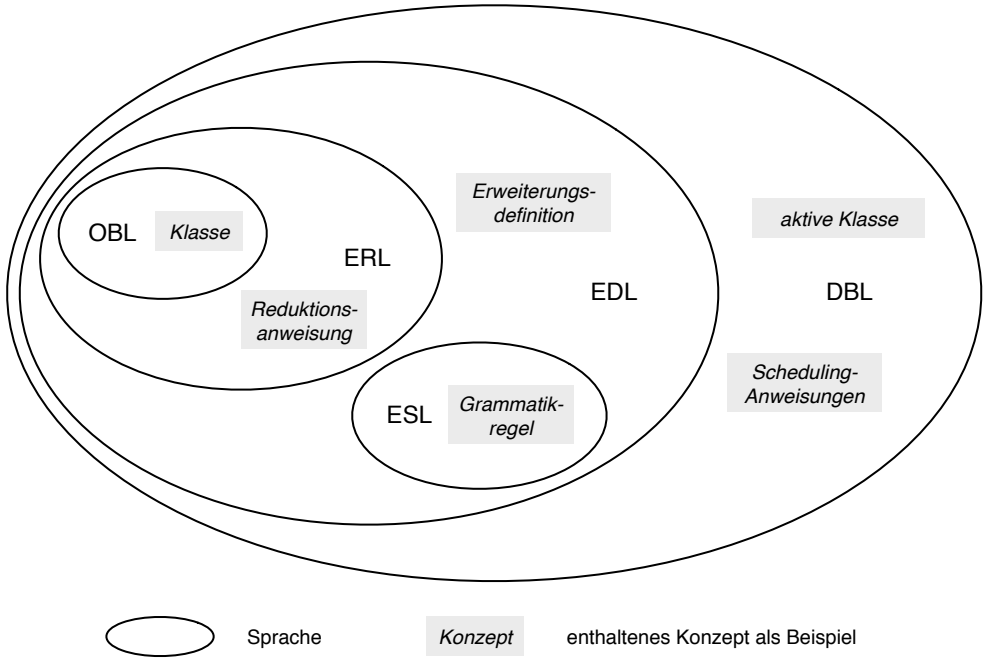

Abbildung 5.2: Teilsprachen von DBL.

```
1 interface Function extends NamedElement, TypedElement, LocalScope {
2   EList getParameters();
3   boolean isStatic();
4   boolean isAbstract();
5 }
```

Listing 5.1: Beispiel für die Darstellung einer OBL-Metaklasse als OBL-Interface.

Realisierung mit Algorithmen zur Syntaxerweiterung und Konzeptreduktion

Die Algorithmen *Syntaxerweiterung* und *Konzeptreduktion* führen Erweiterungsdefinitionen und Erweiterungsinstanzen auf die Basissprache zurück und ermöglichen so die Wiederverwendung von Sprachwerkzeugen der Basissprache für Erweiterungen.

Die Syntaxdefinition wird auf eine Erweiterung der Syntaxdefinition der Basissprache zurückgeführt. Damit wird eine Modellierung mit Erweiterungen unter Wiederverwendung eines Modelleditors für die Basissprache erreicht. Die Definition der Semantik wird dagegen durch eine Konzeptreduktion auf die Konzepte der Basissprache zurückgeführt. Die Basiskonzepte ersetzen dabei die Erweiterungsinstanz. Damit kann ein Modell, das Erweiterungen enthält und vollständig auf Basiskonzepte zurückgeführt wurde, mit dem gleichen Werkzeug wie ein Basissprachenprogramm ausgeführt werden. Beide Algorithmen werden ausführlich in den Abschnitten 5.3 und 5.4 betrachtet.

5.1.2 Implementierung als Framework DMX

Das Framework DMX basiert auf den Technologien Eclipse, EMF [24], TEF [62] und Xtend [37]. Die Komponenten des Ansatzes DMX und ihre Implementierung im Framework DMX werden in Abb. 5.1 dargestellt.

Als Basissprache wird im Framework DMX die in Kapitel 3 vorgestellte Simulationsbasissprache DBL verwendet. Der Quellcode für das Framework ist öffentlich auf Github verfügbar[1]. Das Framework DMX besteht aus einer Reihe von Teilkomponenten in Form von Eclipse-Plugins:

1. einer *Syntaxdefinition* der Simulationsbasissprache DBL,
2. einem *DBL-Java-Transcompiler* für die Übersetzung eines DBL-Modells in ein Java-Programm,
3. einer Implementierung der Syntaxerweiterung als *adaptiver Modelleditor*,
4. einer Implementierung der Konzeptreduktion mit einem *ERL-Java-Transcompiler*
5. und Eclipse-Erweiterungen für Menüs und Aktionen, die Modelleditor und Transcompiler zu einer *Entwicklungsumgebung* integrieren.

Die Syntaxdefinition der Simulationsbasissprache DBL besteht aus einem Ecore-Metamodell und einer Java-Schnittstelle für dieses Metamodell, die auf EMF basiert sowie einer kontextfreien Grammatik für das Metamodell, die in TSL definiert ist und von einem TEF-basierten Modelleditor verwendet wird. Die DBL-Definition umfasst außerdem eine Implementierung für die Auflösung von Bezeichnern in einem DBL-Modell.

Der DBL-Java-Transcompiler übersetzt ein DBL-Modell, das ausschließlich Basiskonzepte enthält, in ein Java-Programm. Java wurde als Zielsprache gewählt, da alle Teilkomponenten des Framework DMX ebenfalls in Java implementiert sind. Durch die Verwendung von Java verringert sich der Aufwand für die Implementierung der Konzeptreduktion, da das Java-basierte Metamodellierungsframework EMF für den Zugriff auf Erweiterungsinstanzen, die in einem DBL-Modell enthalten sind und durch Java-Objekte repräsentiert werden, eingesetzt werden kann.

Der DBL-Java-Transcompiler definiert die Semantik von DBL und dient als Referenzcompiler. Der Transcompiler ist als ausführbare Modell-zu-Code-Transformation in der Metasprache Xtend [37] formuliert. Die Beschreibung erfolgt hier als Übersetzung auf Basis der Konzepte des Metamodells und legt für diese eine Abbildung auf einen Text fest.

Neben Java können Transcompiler für weitere Zielsprachen hinzugefügt werden. Dabei kann die einmal auf der Basis von Java implementierte Konzeptreduktion wiederverwendet werden. Diese übersetzt ein DBL-Modell, das Erweiterungsinstanzen enthält, auf ein DBL-Modell, das ausschließlich aus Basiselementen besteht. Ein solches Basismodell kann als Eingabe für beliebige DBL-Transcompiler dienen.

Besonders interessant ist ein DBL-C++-Transcompiler. Wird dieser nach der in Kapitel 4 beschriebenen Methode implementiert, so kann er für die Erzeugung besonders laufzeiteffizient ausführbarer Simulationsprogramme eingesetzt werden.

[1] https://github.com/ablunk/dmx

Die anderen Teilkomponenten werden in späteren Abschnitten betrachtet. Die Teilkomponente für den adaptiven Modelleditor wird in Abschnitt 5.3 beschrieben. Die Vorstellung des ERL-Java-Transcompilers ist Teil der Konzeptreduktion in Abschnitt 5.4.

5.2 Erweiterungsdefinition

Das Metakonzept für die Erweiterungsdefinition ist Teil der Sprache EDL. Es erlaubt die Angabe eines zu erweiternden Basiskonzeptes durch einen Verweis auf ein OBL-Interface, das die Metaklasse des Basiskonzeptes in OBL repräsentiert. Das Metamodell einer Erweiterungsdefinition wird in Abb. 5.3 gezeigt. Eine Erweiterungsdefinition besteht aus einer Syntaxdefinition in ESL und aus einer Semantikdefinition in ERL.

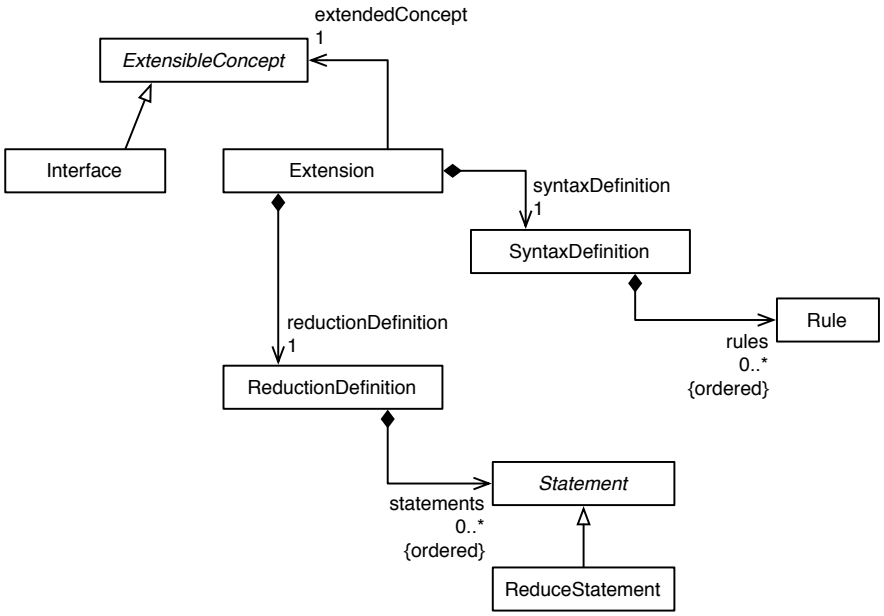

Abbildung 5.3: Metamodell für eine Erweiterungsdefinition.

5.2.1 Beispiel forever-Anweisung

Listing 5.2 zeigt eine Erweiterungsdefinition für eine Forever-Anweisung, die in einer textuellen Notation für EDL angegeben ist. Diese Erweiterung fügt eine neue Anweisung zur Basissprache hinzu, indem das Basiskonzept SimpleStatement erweitert wird.

Die Metaklassen von DBL sind dabei als DBL-Interfaces in DBL selbst in einem speziellen DBL-Modul mit dem Namen dbl vorhanden. Das Modul wird durch die Anweisung `#import "../dbl"` eingebunden. Damit kann das DBL-Interface für die Metaklasse SimpleStatement für die Definition der Erweiterung verwendet werden.

```
#import "../dbl"

module foreverDefinition;

extension Forever extends dbl SimpleStatement {
    // syntax definition
}

// semantics definition
```

Listing 5.2: Beispiel für eine Forever-Erweiterungsdefinition.

5.3 Syntaxerweiterung

Die Syntaxbeschreibungssprache ESL reduziert den Entwicklungsaufwand für Erweiterungen, in dem Ergänzungen des Metamodells und der Grammatik der Basissprache aus einer ESL-Syntaxdefinition abgeleitet werden. Zusätzlich kann die bereits vorhandene Syntaxdefinition für Basiskonzepte in einer ESL-Syntaxdefinition wiederverwendet werden.

Eine Erweiterung stellt dabei bestimmte Voraussetzungen an die Definition der Basiskonzepte. Um die Voraussetzungen und die Ableitung einer Basisergänzung anzugeben, werden das Metamodell und die Grammatik der Basissprache zunächst formal mit einem Metamodell definiert. Dabei wird von konkreten Metasprachen für Metamodelle und Grammatiken abstrahiert. Es wird eine Metamodellierungssprache *Metamodel Language* (MML) und eine Grammatiksprache *Metamodel Grammar Language* (MMGL) eingeführt, die jeweils nur die im vorliegenden Kontext essentiellen Metakonzepte enthalten. Danach wird gezeigt, dass die für die Implementierung verwendeten konkreten Metasprachen Ecore und TSL diese Metakonzepte ebenfalls enthalten.

5.3.1 Metakonzepte für die Syntaxdefinition von Basiskonzepten

Ich definiere die Metakonzepte von MML und MMGL mit einem Metamodell und erkläre danach ihre Semantik, um später Aussagen über die Beschaffenheit von Instanzen dieser Metamodelle für die Erweiterbarkeit zu treffen. Eine Instanz des MML-Metamodells definiert dabei selbst ein MML-Metamodell, während eine Instanz des MMGL-Metamodells eine Grammatik definiert.

Konzepte der Metamodellierungssprache MML

Abb. 5.4 zeigt das Metamodell für die Konzepte von MML in der Notation von UML-Klassendiagrammen. Die Unterschiede zur UML-Notation werden in Abb. 5.5 dargestellt. Die im Metamodell verwendeten Konzepte zur Definition der Konzepte sind genau die Konzepte, die das Metamodell selbst definiert. Somit sind alle Konzepte enthalten, so dass die Metamodellierungssprache MML ihre abstrakte Syntax selbst definieren kann.

Ein MML-Modell besteht aus Metaklassen (MetaClass), die

- einen Namen besitzen,
- abstrakt oder nicht abstrakt sind,
- Metaobjektattribute besitzen (attributes)
- und von anderen Metaklassen erben (superClasses).

Ein Metaobjektattribut (MetaObjectAttribute) besitzt

- einen Namen,
- eine Referenz- oder eine Kompositionssemantik für Werte (allocationKind),
- eine Kardinalität für die Anzahl der Attributwerte (collectionKind) mit den möglichen Ausprägungen: genau ein Wert (exactlyOne), höchstens ein Wert (atMostOne) und eine Liste beliebig vieler Werte (list)
- und einen Typ für die Attributwerte, der eine Metaklasse, ein Elementartyp (PrimitiveTyp) oder ein Aufzählungstyp (Enumeration) sein kann.

Elementartypen (PrimitiveType) gibt es nur in den Ausprägungen Integer, String und Boolean. Ein Aufzählungstyp (Enumeration) definiert eine Liste von Aufzählungswerten (EnumerationLiteral).

Beispiel für ein MML-Metamodell

Als Beispiel für ein MML-Metamodell wird in Abb. 5.6 die abstrakte Syntax einer DBL-Funktion gezeigt. Eine Funktion kann eine statische Klassenmethode (static = true) oder eine Objektmethode sein (static = false). Sie kann abstrakt oder nicht abstrakt sein. Eine Funktion hat einen Namen, einen Rückgabetyp (TypedElement) und eine Liste von Parametern. Sie definiert einen lokalen Gültigkeitsbereich (LocalScope), der für die Anweisungen der Funktion (statements) relevant ist.

Vergleich mit den Metakonzepten von Ecore

Ecore ist eine Metamodellierungssprache, die die beschriebenen Metakonzepte enthält und deshalb für die Implementierung im Framework DMX verwendet werden kann. Der Nachweis wird über die Angabe einer Spezialisierung zwischen dem Ecore-Metamodell und dem MML-Metamodell geführt, die in Abb. 5.7 und 5.8 dargestellt wird. Die Attribute der MML-Klassen werden als Ableitungen aus den Attributen der Ecore-Metaklassen in der *Object Constraint Language* (OCL) [51] in den Listings 5.3 und 5.4 angegeben.

Konzepte der Grammatiksprache MMGL

Eine Instanz der Grammatiksprache für Metamodelle definiert eine Grammatik mit einer Beziehung zu einem MML-Metamodell, die eine Wortfolge entsprechend der Grammatikregeln konsumiert und auf eine Instanz des MML-Metamodells abbildet. Diese besondere Grammatik kann deshalb als Metamodellgrammatik bezeichnet werden.

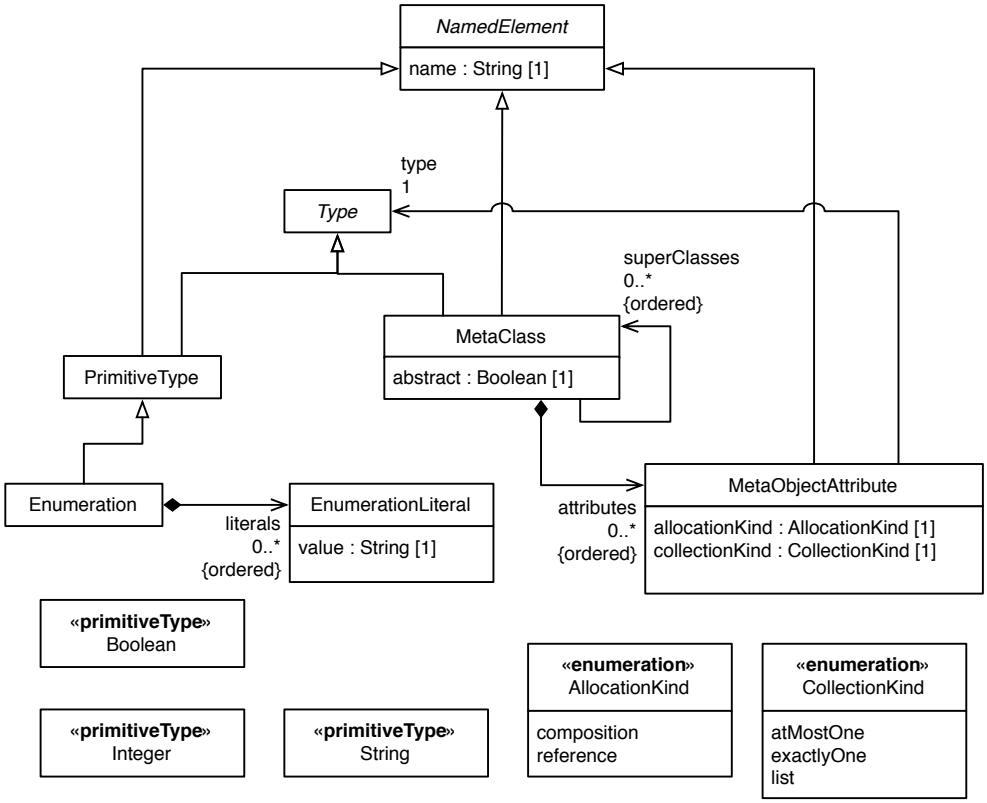

Abbildung 5.4: Metamodell für MML in UML-Notation.

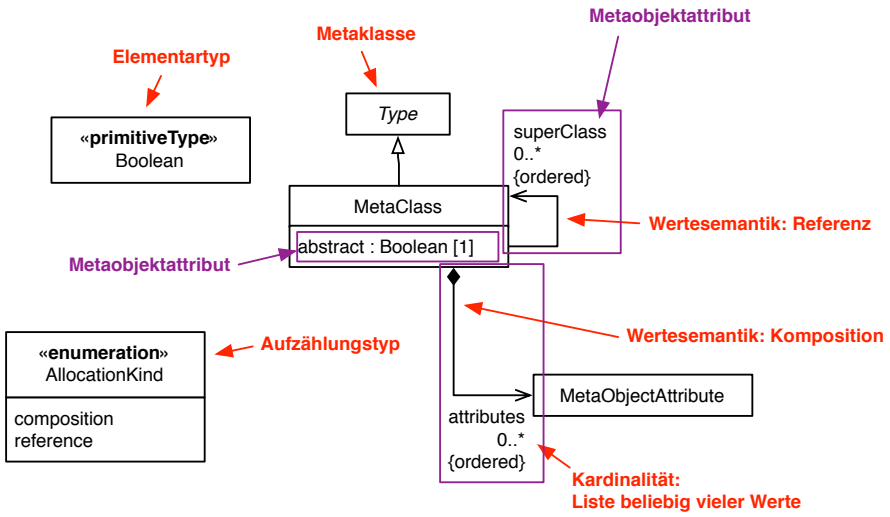

Abbildung 5.5: Unterschiede der Notation von MML-Modellen zur UML-Notation.

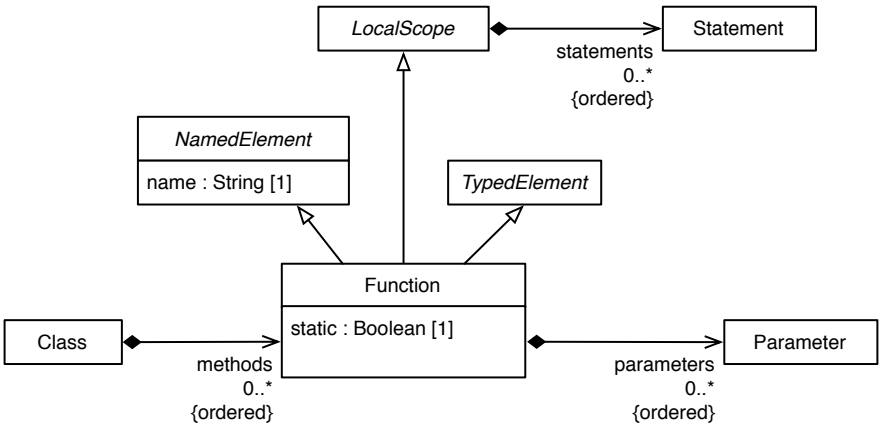

Abbildung 5.6: DBL-Funktion als MML-Metamodell.

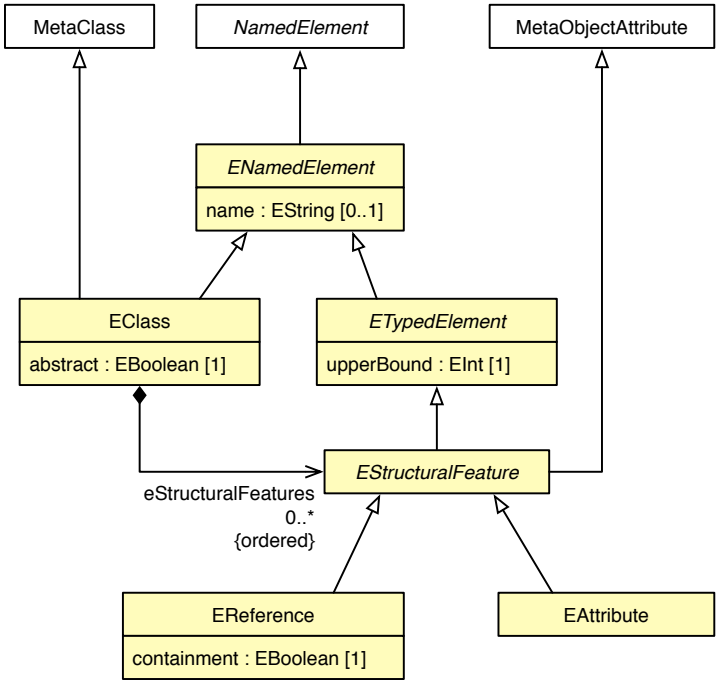

Abbildung 5.7: Ecore als Spezialisierung von MML (Teil 1).

```
1 context EClass::attributes : MetaAttribute
2 derive: eStructuralFeatures
3
4 context EClass::superClasses : MetaClass
5 derive: eSuperTypes
```

Listing 5.3: Ableitung der Attribute der Klasse MetaClass im Kontext der Klasse EClass mit OCL.

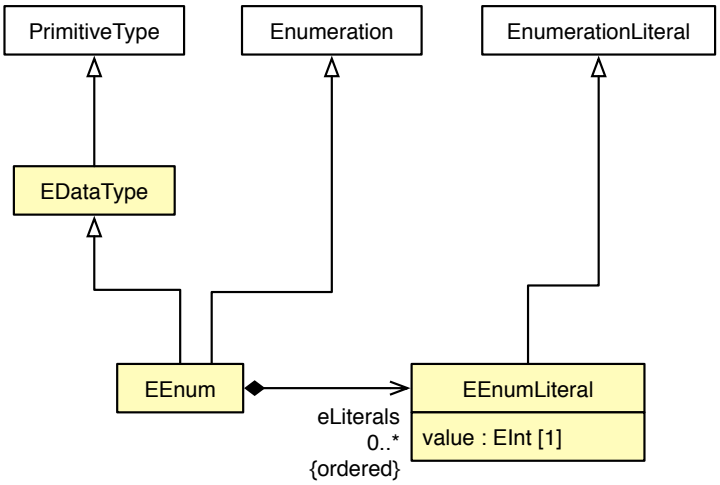

Abbildung 5.8: Ecore als Spezialisierung von MML (Teil 2).

```
1  context EStructrualFeature::allocationKind : AllocationKind
2  derive:
3    if self.oclIsTypeOf(EAttribute) or self.oclAsType(EReference).containment
4    then AllocationKind::composition
5    else AllocationKind::reference
6    endif
7
8  context EStructrualFeature::collectionKind : CollectionKind
9  derive:
10   if upperBound = 1 then CollectionKind::one else CollectionKind::list endif
```

Listing 5.4: Ableitung der Attribute der Klasse MetaObjectAttribute im Kontext der Klasse EStructrualFeature mit OCL.

Das MMGL-Metamodell wird in Abb. 5.9 dargestellt. Eine MMGL-Grammatik besteht aus einer Syntaxdefinition, die eine Liste von Regeln (rules) enthält und eine dieser Regeln als Startregel (start) festlegt.

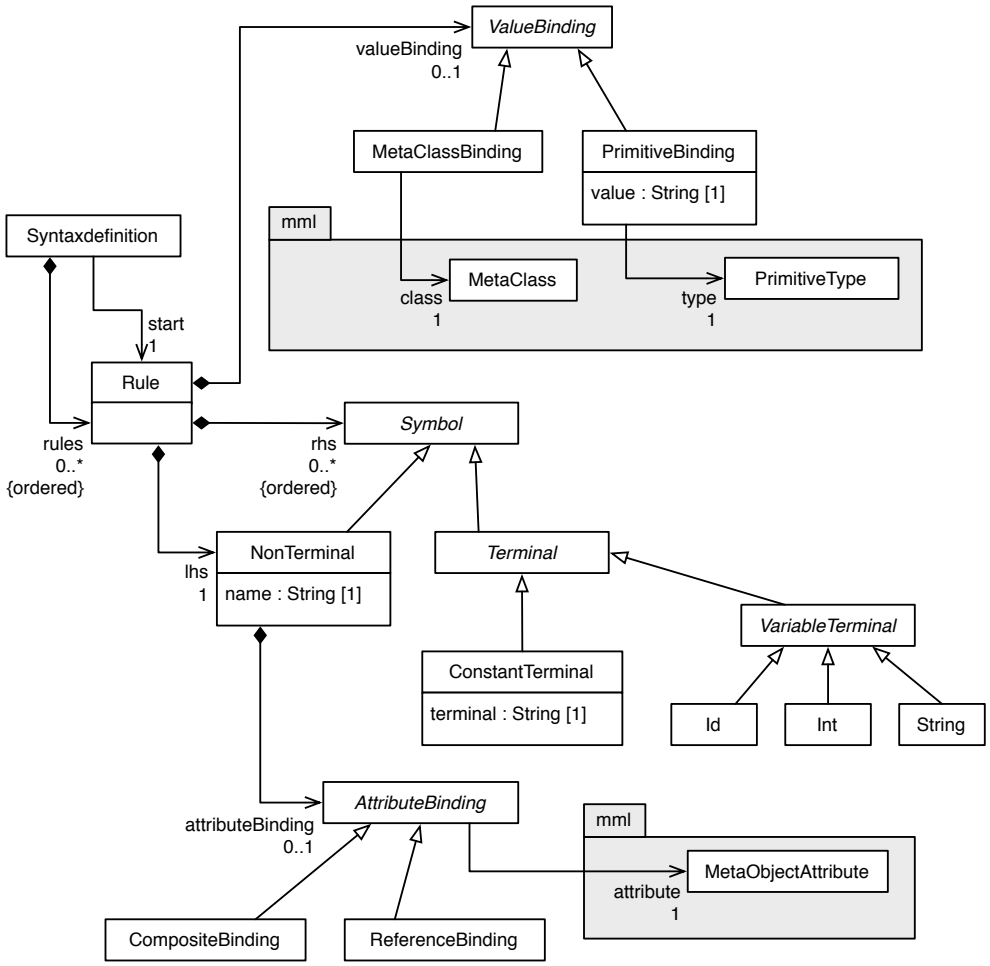

Abbildung 5.9: MML-Metamodell für MMGL.

Eine Regel leitet ein Nichtterminalsymbol auf der linken Seite (lhs), zu einer Folge von Nichtterminal- und Terminalsymbolen auf der rechten Seiten (rhs) ab. Diese Definition entspricht der Definition von Regeln in einer kontextfreien Grammatik. Zusätzlich können Syntaxattribute für ein Nichtterminalsymbol definiert werden. Die möglichen Syntaxattribute sind: i) Bindung an eine Metaklasse, ii) Bindung an einen Elementarwert und iii) Bindung an ein Metaobjektattribut als Attributbindung.

Eine Regel r kann über das Nichtterminalsymbol auf ihrer linken Seite eine Bindung an eine Metaklasse (MetaClassBinding) oder eine Bindung an einen Elementarwert (PrimitiveBinding) definieren. Wird die Regel abgeleitet, so wird ein Objekt der entsprechenden Metaklasse oder ein entsprechender Elementarwert erzeugt. Das Metaobjektattribut a, in dem dieser Wert gespeichert wird, muss über eine Attribut-

bindung (AttributeBinding), als Teil einer anderen Regel r', angegeben werden. Dazu muss das Nichtterminalsymbol der linken Seite der Regel r auf der rechten Seite der Regel r' unter Angabe einer Attributbindung an a verwendet werden. Auf diese Weise entsteht durch eine Ableitung der Regeln eine Instanz des Metamodells.

Eine Attributbindung kann nur für ein Nichtterminalsymbol auf der rechten Seite einer Regel definiert werden. Die Attributbindung muss dabei festlegen, ob der Wert, der bei der Ableitung des Nichtterminalsymbols erzeugt wird, im angegebenen Metaobjektattribut gespeichert werden soll (CompositeBinding) oder ob lediglich eine Referenz auf das angegebene Objekt zu speichern ist (ReferenceBinding). Das Nichtterminalsymbol einer Referenzbindung muss dabei eine Ableitung auf einen Bezeichner definieren, der eine Attributbindung an einem Metaobjektattribut name festlegt. So wird bei Ableitung eines entsprechenden Bezeichners, an der Stelle der Referenzbindung, eine Objektreferenz auf das Objekt erzeugt, dessen name-Attribut den Wert des Bezeichners enthält.

Jedes Nichtterminalsymbol mit einer Attributbindung muss konsistent zum Metaobjektattribut der gebundenen Metaklasse definiert werden. Wird eine Wertbindung verwendet, so muss das Metaobjektattribut eine Kompositionssemantik festlegen. Wird dagegen eine Referenzbindung benutzt, so muss das Metaobjektattribut eine Referenzsemantik definieren.

Ein Nichtterminalsymbol besitzt einen Namen und kann über diesen Namen mehrfach in verschiedenen Regeln verwendet werden. Ein Terminalsymbol kann eine feste (ConstantTerminal) oder eine variable Zeichenfolge sein (VariableTerminal). Eine variable Zeichenfolge wird bei einer kontextfreien Grammatik über einen regulären Ausdruck auf eine bestimmte Zeichenfolge festgelegt. Die vorliegende Grammatik ist vereinfachend auf die variablen Zeichenfolgen Zeichenkette (String), ganze Zahl (Int) und Bezeichner (Id) beschränkt.

Die Textual Syntax Language (TSL), die im Framework TEF verwendet wird, ist eine Grammatiksprache für Metamodelle, die die Konzepte der Sprache MMGL enthält und damit für die Implementierung der Syntaxerweiterung eingesetzt werden kann.

Beispiel für eine Metamodellgrammatik

Abb. 5.10 zeigt ein Beispiel für ein Binding zwischen einer Metamodellgrammatik und einem Metamodell. Die dargestellte Grammatik erzeugt für die angegebene Wortfolge ein Modell als Instanz des Metamodells.

Die MMGL-Grammatik ist dabei in einer Notation angegeben, die eine prägnante Darstellung erlaubt. Es gibt die Regeln P, M, ManyM, F und R. Dabei ist P die Startregel. Eine Regel definiert links vom Schlüsselwort -> ein Nichtterminalsymbol und rechts davon eine Folge von Nichtterminal- und Terminalsymbolen, die mit einem Semikolon ; abgeschlossen wird.

Die Bindung an eine Metaklasse wird mit dem Schlüsselwort object für ein Nichtterminalsymbol auf der linken Seite definiert. Eine Bindung an einen Elementarwert wird dagegen mit dem Schlüsselwort const festgelegt. Auf das Schlüsselwort const folgt der Wert selbst als Zeichenfolge und der Elemtartyp des Wertes.

Auf der rechten Seite einer Regel befinden sich außerdem Attributbindungen. Eine Wertbindung wird mit dem Schlüsselwort composite und eine Referenzbindung wird

mit dem Schlüsselwort reference angegeben.

5.3.2 Metakonzepte für die Syntaxdefinition von Erweiterungskonzepten

Das Ziel bei der Auswahl der passenden Metakonzepte ist die Zurückführung einer Syntaxdefinition für eine Erweiterung auf die Syntaxdefinition der Basissprache. Zusätzlich müssen die Metakonzepte in der Basissprache enthalten sein, so dass eine Erweiterung in der Basissprache definiert werden kann. Da die Metakonzepte nur für die Syntaxdefinition geeignet sind, fasse ich diese in der Teilsprache *Extension Syntax Language* (ESL) zusammen. Die Sprache ESL muss in der Basissprache enthalten sein, damit Erweiterungen definiert werden können.

Die Syntaxdefinition von Erweiterungen könnte mit den gleichen Metakonzepten wie die Syntaxdefinition der Basissprachen erfolgen. Dabei müsste ein Metamodell und eine Metamodellgrammatik für die Erweiterung definiert werden, die das Metamodell und die Metamodellgrammatik der Basissprache erweitern. Das Ziel der Arbeit ist es jedoch den Aufwand für die Entwicklung einer DSL zu verringern. Eine Aufwandsreduktion kann durch die Ableitung eines Metamodells und einer Metamodellgrammatik aus einer kombinierten Syntaxdefinition erreicht werden. Diese Ableitung ist möglich, da die Sprachen zur Definition von Metamodell und Metamodellgrammatik ähnliche Konzepte enthalten. Diese Konzepte werden in ESL zusammengefasst, so dass eine ESL-Syntaxdefinition als Grundlage für eine Ableitung dienen kann. Die Beschreibung der Ableitung ist Teil des Abschnittes 5.3.4.

Den Ausgangspunkt einer Syntaxdefinition bildet das zu erweiternde Basiskonzept, das in der Erweiterungsdefinition angegeben wird. Für den vorliegenden Abschnitt treffe ich die Annahme, dass über das zu erweiternde Basiskonzept sowohl die Metaklasse als auch eine Grammatikregel für das Basiskonzept identifiziert werden. Diese Annahme trifft nur unter bestimmten Voraussetzungen zu, die im Abschnitt 5.3.3 erläutert werden.

Die im Abschnitt 5.3.1 vorgestellten Metakonzepte der Sprache für Metamodellgrammatiken bilden die Grundlage für ESL. Die Metakonzepte werden wie folgt angepasst:

1. eine Regel definiert keine Bindung an eine Metaklasse,
2. ein Symbolattribut enthält statt eines Verweises auf ein Metaobjektattribut eine Definition für ein Metaobjektattribut durch die Angabe eines Bezeichners,
3. ein Symbolattribut hat zusätzlich eine Eigenschaft für die Kardinalität der Wertemenge des Attributes mit entweder höchstens einem Wert oder einer Liste beliebig vieler Werte und
4. sowohl variable als auch konstante Terminalsymbole können ein Syntaxattribut definieren.

Das Metamodell für die Regeln in einer ESL-Syntaxdefinition wird in Abb. 5.11 dargestellt. Eine Regel ist ein Nichtterminalsymbol und definiert so ihre linke Seite. Sie besteht auf der rechten Seite aus einer Folge von Symbolausdrücken (SyntaxSymbolExpression), die auf ein Nichtterminalsymbol verweisen oder ein Terminalsymbol enthalten. Ein Symbolausdruck kann ein Syntaxattribut definieren, das einen

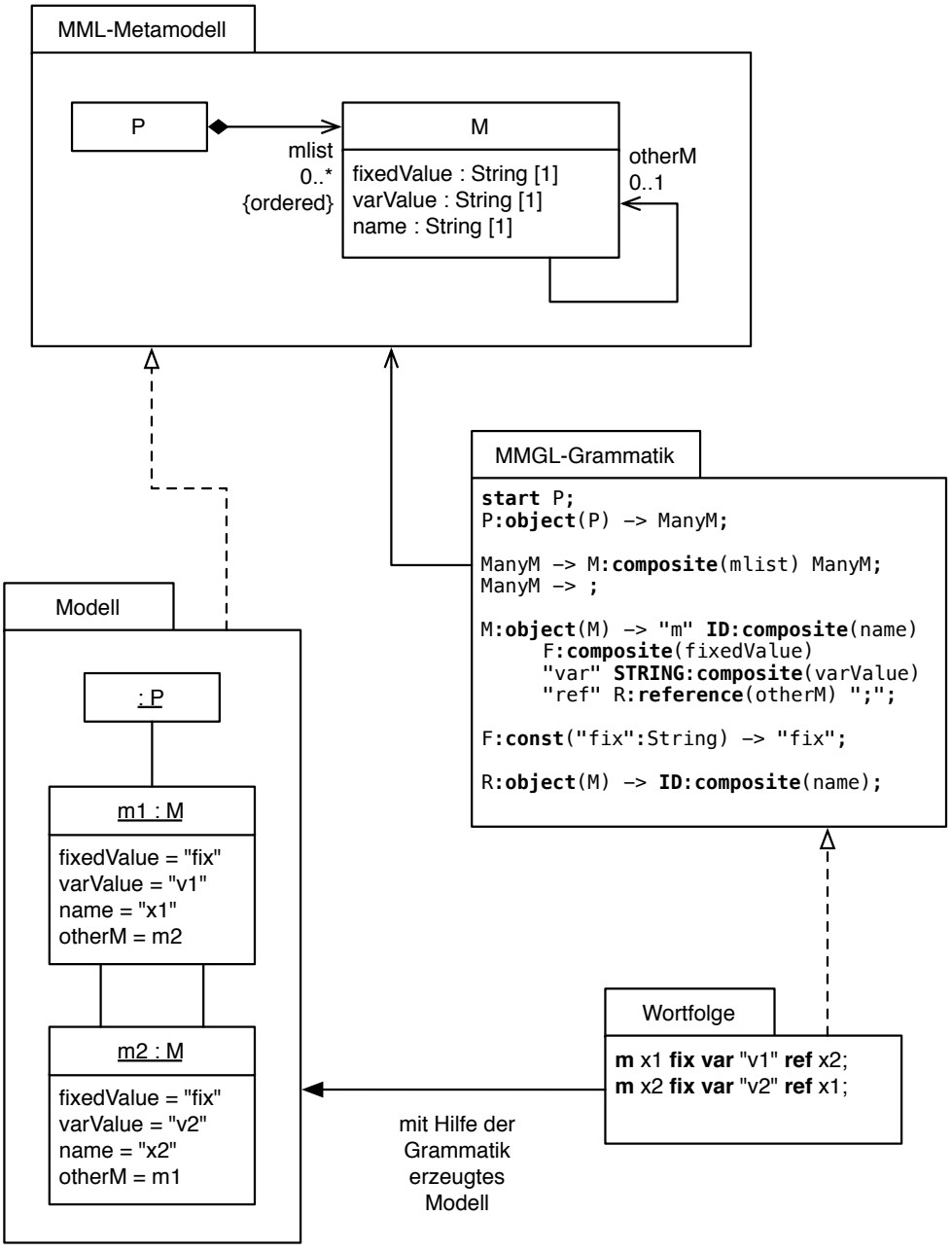

Abbildung 5.10: Beispiel für eine Metamodellgrammatik mit einer Bindung an ein Metamodell.

Namen, eine Semantik für die Wertzuweisung, eine Kardinalität für die Wertemenge und eine Strategie für die Auflösung von Bezeichnern bei einer Referenzsemantik festlegt.

Nichtterminalsymbole und Syntaxattribute dienen als Grundlage für die Ableitung von Metaklassen und Metaobjektattributen. Mit einem Syntaxattribut wird daher ein Metaobjektattribut für das Nichtterminalsymbol definiert.

Abb. 5.12 zeigt das Metamodell für die Syntaxsymbole, die auf der rechten Seite einer Regel vorkommen können. Die Terminalsymbole sind auch hier auf die Symbole Bezeichner, ganze Zahl und Zeichenkette beschränkt.

Die Nichtterminalsymbole umfassen dagegen neben Regeln auch erweiterbare Konzepte, zu denen Basiskonzepte (repräsentiert durch ein DBL-Interface für eine Basismetaklasse) und Erweiterungen gehören. Die Hinzunahme von Symbolen für Metaklassen erlaubt die Wiederverwendung der Syntaxdefinition von Basiskonzepten in Erweiterungen.

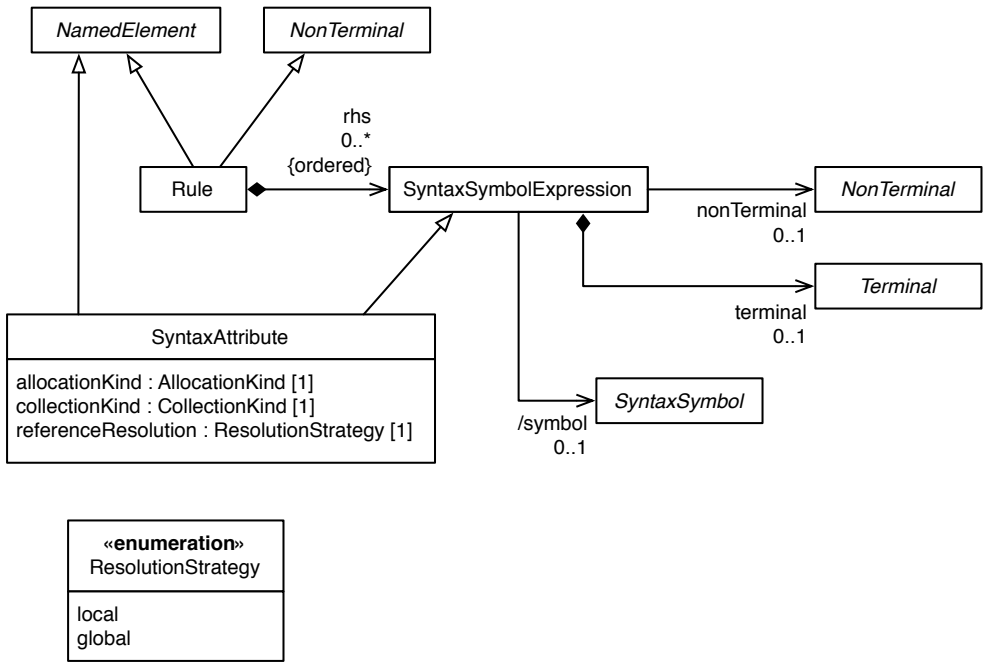

Abbildung 5.11: Metamodell für die Regeln in einer ESL-Syntaxdefinition.

Eine Notation für ESL

Beispiel 1 – ESL-Syntaxdefinition für eine abstrakte M-Anweisung

Listing 5.5 zeigt eine zur Metamodellgrammatik in Abb. 5.10 äquivalente Syntaxdefinition in ESL, die in Listing 5.6 verwendet wird.

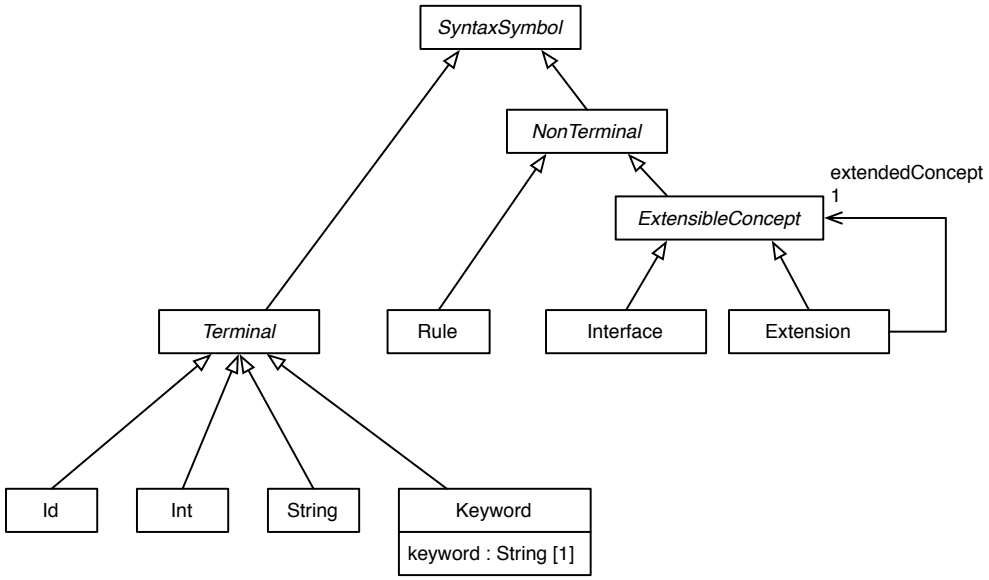

Abbildung 5.12: Metamodell für die Syntaxsymbole in einer ESL-Syntaxdefinition.

```
#import "../dbl"

module mDefinition;

extension M extends SimpleStatement {
  start P;
  P -> ManyM;

  ManyM -> M:list mlist ManyM;
  ManyM -> ;

  M -> "m" ID:name
    "fix":fixedValue
    "var" STRING:varValue
    "ref" M:$otherM ";";
}
```

Listing 5.5: Beispiel für eine Erweiterungsdefinition mit einer Syntaxdefinition, die äquivalent zur Metamodellgrammatik in Abb. 5.10 ist.

```
#import "../mDefinition"

void main() {
  m x1 fix var "v1" ref x2;
  m x2 fix var "v2" ref x1;
}
```

Listing 5.6: Beispiel für eine Erweiterungsinstanz entsprechend der Erweiterungsdefinition aus Listing 5.5.

Beispiel 2 – ESL-Syntaxdefinition für eine forever-Anweisung

Listing 5.7 zeigt eine Erweiterungsdefinition mit einer Syntaxdefinition für eine Forever-Anweisung, die in einer textuellen Notation für EDL angegeben ist. Diese Erweiterung fügt eine neue Anweisung zur Basissprache hinzu, indem die Metaklasse und die Grammatikregel für das Basiskonzept SimpleStatement erweitert werden.

```
1 #import "../dbl"
2
3 module foreverDefinition;
4
5 extension Forever extends dbl SimpleStatement {
6   start Forever;
7   Forever -> "forever" body:LocalScopeStatement;
8 }
```

Listing 5.7: Beispiel für eine Erweiterungsdefinition mit einer Syntaxdefinition für eine Forever-Anweisung.

Die Syntaxdefinition legt als Startsymbol das Nichtterminalsymbol Forever fest. Die Regel für das Nichtterminalsymbol Forever definiert eine Ableitung auf das Schlüsselwort forever, gefolgt vom einen Nichtterminalsymbol für die Metaklasse des Basiskonzeptes LocalScopeStatement mit einem Syntaxattribut body, das eine Instanz von LocalScopeStatement für eine zugehörige abgeleitete Grammatikregel für ein LocalScopeStatement enthält.

Ein Beispiel für eine Forever-Erweiterungsinstanz ist in Listing 5.8 dargestellt. Die aktive Klasse ArrivalOfA beschreibt einen einfachen Ankunftsprozess, in dem bis zum Ende der Simulation, im Abstand von einer Zeiteinheit, aktive Objekte der Klasse A erzeugt und aktiviert werden.

```
1  #import "foreverDefinition"
2  #import "a"
3
4  module foreverUse;
5
6  active class ArrivalOfA {
7    actions {
8      forever {
9        activate new A();
10       advance 1;
11     }
12   }
13 }
14
15 void main() {
16   activate new ArrivalOfA();
17   advance 60;
18 }
```

Listing 5.8: Beispiel für eine Forever-Erweiterungsinstanz.

5.3.3 Voraussetzungen an die Basissyntaxdefinition

Den Ausgangspunkt einer Syntaxdefinition bildet die Angabe eines zu erweiternden Basiskonzeptes in einer Erweiterungsdefinition. Die Angabe erfolgt dabei über einen Verweis auf die Metaklasse, die die abstrakte Syntax des Basiskonzeptes definiert. Die Metaklasse und die Grammatikregel für das Basiskonzept müssen bestimmte Voraussetzungen erfüllen damit das Basiskonzept erweiterbar ist.

Da das Metamodell und die Grammatik der Basissprache mit einem Metamodell definiert sind, können die Voraussetzungen, die für beide Metamodellinstanzen gelten müssen, in OCL angegeben werden. Listing 5.9 definiert die Erweiterbarkeit eines Basiskonzeptes durch Ableitung des Attributes extensible der Metaklasse MetaClass.

```
1  context MetaClass::extensible : Boolean
2  derive:
3    Rule.allInstances()->exists(r | r.lhs.name = self.name and r.valueBinding = null)
4    and self.abstract = false
5    and MetaObjectAttribute.allInstances()->exists(a | a.type = self
6        and a.allocationKind = AllocationKind::composition)
```

Listing 5.9: Ableitung für das Attribut extensible in OCL.

Voraussetzung 1: Für eine Metaklasse m gibt es eine gleichnamige Regel ohne Bindung an m

Der erste Teil des Ableitungsausdrucks legt fest, dass eine Regel existieren muss, die keine Bindung an eine Metaklasse oder an einen Elementarwert definiert und den gleichen Namen wie die Metaklasse besitzt. Diese Teilbedingung löst das Problem, das für die Basisregel noch nicht feststehen darf, welche Metaklasse zu instanziieren ist. Eine Erweiterung führt nämlich eine Alternative für die Basisregel ein, bei deren Ableitung eine Spezialisierung der Basismetaklasse zu instanziieren ist.

Folglich darf in der Basisgrammatik die Bindung an die Basismetaklasse erst in einer Alternative der Basisregel erfolgen. Das Problem der notwendigen Identifikation von Basismetaklasse und Basisregel für die Einführung einer Erweiterung wird über die Namensgleichheit der beiden Elemente erreicht.

Listing 5.10 zeigt als Beispiel die erweiterbare Definition des Basiskonzeptes Funktion.

```
1  Function -> BoundFunction;
2  BoundFunction:object(Function) -> BoundFunctionWithModifiers;
3
4  BoundFunctionWithModifiers -> FunctionStatic;
5  BoundFunctionWithModifiers -> FunctionAfterModifiers;
6
7  FunctionStatic -> Static:composite(static) FunctionAfterModifiers;
8
9  FunctionAfterModifiers -> TypedElementParts NamedElementName
10   "(" Parameter:composite(parameters) MoreParameters ")"
11   FunctionBody;
12 FunctionBody -> "{" ManyStatements "}";
13 FunctionBody -> ";";
14
15 MoreParameters -> "," Parameter:composite(parameters) MoreParameters;
```

```
16 MoreParameters -> ;
17
18 ManyStatements -> Statement:composite(statements) ManyStatements;
19 ManyStatements -> ;
```

Listing 5.10: Beispiel für die erweiterbare Definition des Basiskonzeptes Funktion.

Voraussetzung 2: Die Metaklasse m ist nicht abstrakt

Die zweite Voraussetzung wird durch den zweiten Teil des Ableitungsausdrucks festgelegt. Hierbei handelt es sich um eine technische Voraussetzung, die durch das für die Implementierung verwendete Metamodellierungs-Framework EMF vorgegeben ist. Für die Programmierung mit Modellen wird ein Ecore-Metamodell durch statische Code-Generierung auf Java-Klassen abgebildet. Eine Erweiterung der Basissprache erfolgt jedoch immer dynamisch während der Ausführung des adaptiven Modelleditors. Deshalb ist für die Metaklasse einer Erweiterung keine Java-Klasse für die Instanziierung vorhanden.

Stattdessen muss die Java-Klasse des erweiterten Basiskonzeptes verwendet werden, um sicherzustellen, dass Sprachwerkzeuge, die auf der Grundlage des Basiskonzeptes implementiert sind, auch ein Erweiterungskonzept verarbeiten können. Dazu darf die entsprechende Metaklasse jedoch nicht von vornherein als abstrakt definiert sein.

Voraussetzung 3: Es gibt ein Eltern-Konzept

Der dritte Teil des Ableitungsausdrucks legt fest, dass jede Erweiterungsinstanz im Objektbaum des Basisprogramms gespeichert werden kann. Damit das Basiskonzept einer Metaklasse m erweiterbar ist, muss es ein Attribut mit einer Kompositionssemantik in einer anderen Metaklasse geben, das vom Typ der Basismetaklasse m ist.

Die Bedingung kann am Beispiel der Funktion in Abb. 5.6 veranschaulicht werden. Eine Funktion ist in einer Instanz der Metaklasse Class enthalten und damit ist Voraussetzung 3 erfüllt.

5.3.4 Abbildung auf die Basissyntaxdefinition

Dieser Abschnitt zeigt, dass aus einer ESL-Syntaxdefinition sowohl ein Metamodell, als auch eine Metamodellgrammatik als Erweiterung der Basissyntaxdefinition abgeleitet werden können. Die Ableitung wird allgemein beschrieben und mit der Darstellung in Abb. 5.13 veranschaulicht.

Gegeben sei eine Erweiterungsdefinition E für ein erweiterbares Basiskonzept K, das durch eine Metaklasse K und eine Grammatikregel K mit dem gleichen Namen K definiert ist.

Der erste Schritt ist die Einführung einer neuen Metaklasse E und einer neuen Grammatikregel E. Die Metaklasse E spezialisiert die Metaklasse K. Die Grammatikregel E für die Erweiterung wird als Alternative für die Grammatikregel K des Basiskonzeptes festgelegt. Dazu definiert die neue Regel eine Ableitung für das Nichtterminalsymbol K des Basiskonzeptes auf das Nichtterminalsymbol E der Regel der

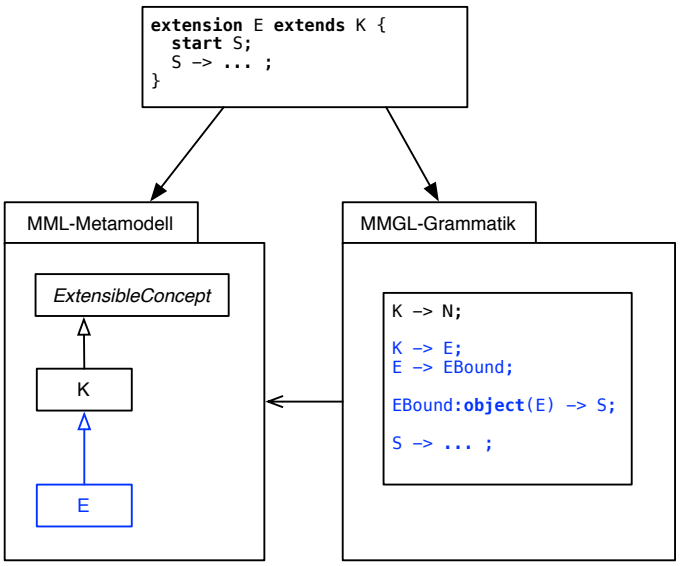

Abbildung 5.13: Einführung einer Metaklasse und einer Regel für eine Erweiterung.

Erweiterung. Des Weiteren wird eine Grammatikregel $EBound$ als Alternative für die Regel E hinzugefügt.

Die Regel $EBound$ definiert eine Ableitung auf das Startsymbol der Syntaxdefinition der Erweiterung S. Das Nichtterminalsymbol $EBound$ besitzt ein Syntaxattribut mit einer Bindung an die Metaklasse E. Wird das Startsymbol S abgeleitet, so wird eine Erweiterungsinstanz als Objekt der Metaklasse E erzeugt.

Die Metaklasse und die Regel verwenden den gleichen Namen. Außerdem besitzt die Regel kein Syntaxattribut mit einer Bindung an die Metaklasse. Damit ist nach Voraussetzung 1 auch die Erweiterung einer Erweiterung möglich.

Abbildung von Regeln

Der nächste Schritte ist die Abbildung aller ESL-Regeln $S \rightarrow \ldots$, die auf der linken Seite das Startsymbol S besitzen. Da die Abbildung für alle Regeln einer ESL-Syntaxdefinition nach der gleichen Vorschrift erfolgt, beschreibe ich die Abbildung allgemein für alle Regeln, die eine Ableitung eines Nichtterminalsymbols R auf eine Folge von Nichtterminal- und Terminalsymbolen definieren.

Folgende Aktionen werden für jede ESL-Regel durchgeführt. Dabei wird angenommen, dass jede ESL-Regel eine Ableitung eines Nichtterminalsymbols R auf eine Folge von Nichtterminal- und Terminalsymbolen $(Y_i)_{i=1,\ldots,n}$ definiert.

- Hinzufügen einer MMGL-Regel $R \rightarrow (Y_i)_{i=1,\ldots,n}$ mit Nichtterminalsymbolen Y_i
- für jedes Nichtterminalsymbol Y_i mit einem Syntaxattribut a_i:
  - Hinzufügen einer MML-Klasse Y_i

  - Hinzufügen eines MML-Attributes a_i in der MML-Metaklasse R, falls kein MML-Attribut a_i in der MML-Metaklasse K oder in einer Oberklasse vorhanden ist
  - Setzen von Eigenschaften für das hinzugefügte MML-Attribut a_i:
    * Metaklasse Y_i als Typ
    * Kompositionssemantik falls a_i keine Referenzsemantik definiert
    * Referenzsemantik falls a_i eine Referenzsemantik definiert
    * Kardinalität als genau ein Wert falls a_i keine Liste definiert, sonst Kardinalität als Liste beliebig vieler Werte
  - falls a_i eine Referenzsemantik definiert:
    * Hinzufügen einer MMGL-Regel V_i mit einer Bindung an die Metaklasse Y_i und einer Ableitung des Nichtterminalsymbols V_i auf das Terminalsymbol Bezeichner mit einer Attributbindung an das MML-Attribut name als Komposition
    * Abbildung von Y_i auf das MMGL-Nichtterminalsymbol V_i in der Regel R mit einer Attributbindung für das MML-Attribut a_i als Referenz
  - falls a_i eine Kompositionssemantik definiert: Abbildung von Y_i auf das MMGL-Nichtterminalsymbol Y_i in der Regel R mit einer Attributbindung für das MML-Attribut a_i als Komposition

- für jedes Terminalsymbol Y_i mit einem Syntaxattribut a_i:
  - bei einem variablem Terminalsymbol vom Typ Zeichenkette oder ganze Zahl:
    * Hinzufügen eines MML-Attributes a_i in der MML-Metaklasse R nach der Vorschrift für Nichtterminalsymbole mit den Eigenschaften:
      · Elementartyp String bei Zeichenkette als Typ
      · Elementartyp Int bei ganzer Zahl als Typ
      · Kompositionssemantik
    * Abbildung von Y_i auf das entsprechende MMGL-Terminalsymbol mit einer Attributbindung für das MML-Attribut a_i als Komposition
  - bei einem festen Terminalsymbol:
    * Hinzufügen eines MML-Attributes a_i in der MML-Metaklasse R nach der Vorschrift für Nichtterminalsymbole mit den Eigenschaften:
      · Elementartyp Boolean als Typ
      · Kompositionssemantik
    * Abbildung von Y_i auf das MMGL-Nichtterminalsymbol C_i mit einer Attributbindung für das MML-Attribut a_i
    * Hinzufügen einer Regel für die Ableitung des Nichtterminalsymbols C_i mit einer Bindung an den Elementartyp Boolean und dem Wert true auf das Terminalsymbol Y_i

  - bei einem variablem Terminalsymbol vom Typ Bezeichner: Abbildung von Y_i auf das MMGL-Terminalsymbol ID mit einer Attributbindung für das MML-Attribut name

- für jedes Nichtterminalsymbol Y_i ohne Syntaxattribut: Abbildung der Ableitungen für Y_i nach der allgemeinen Vorschrift unter Verwendung der MML-Metaklasse R anstelle der Einführung einer neuen Metaklasse Y_i
- für jedes Terminalsymbol Y_i ohne Syntaxattribut: Abbildung von Y_i auf das MMGL-Terminalsymbol Y_i in der MMGL-Regel R

Merkmale der Abbildung

Für jedes Nichtterminalsymbol mit einem Syntaxattribut wird eine passende Metaklasse automatisch erzeugt. Die Definition der Wertesemantik und Kardinalität von Attributen erfolgt dabei einmalig in der ESL-Syntaxdefinition, aus der die entsprechenden MML-Attribute automatisch abgeleitet werden. Entlang der Nichtterminalsymbole mit Syntaxattributen entsteht ein Objektbaum, bei dem die Kind-Objekte in einem Eltern-Objekt entsprechend der definierten Kompositionssemantik enthalten sind.

Wird ein Nichtterminalsymbol N ohne ein Syntaxattribut in einer Ableitung eines Nichtterminalsymbols A verwendet, so wird keine neue Metaklasse für N und kein Attribut im Kontext von A hinzugefügt. Zusätzlich werden die Nichtterminalsymbole mit Syntaxattributen, für die N eine Ableitung definiert, im Kontext von A verarbeitet. Damit erhält die Metaklasse für A die entsprechenden Metaobjektattribute. Auf diese Weise können Objektkompositionen vermieden werden. Des Weiteren lassen sich Wiederholungen von Zeichenfolgen beschreiben, die in einem Attribut vom Typ einer Liste im Kontext eines bestimmten Objektes enthalten sind.

Für Syntaxattribute mit einem Bezeichner als Terminalsymbol wird kein Metaobjektattribut erzeugt. Die Basismetaklasse jedes erweiterbaren Sprachkonzeptes ist als Spezialisierung der Metaklasse NamedElement definiert. Somit besitzt jedes erweiterbare Sprachkonzept und jede Erweiterung ein name-Attribut. Auf diese Weise kann für Erweiterungen eine allgemeine Namensauflösung verwendet werden, die auf Basis von NamedElement als Teil der Basissprache definiert ist.

Die Wiederverwendung von Metaobjektattributen einer Basismetaklasse erlaubt es Funktionen von Basissprachwerkzeugen für Erweiterungen zu verwenden, die für das entsprechende Metaobjektattribut der Basismetaklasse implementiert ist. Ein Beispiel ist die Wiederverwendung der für bestimmte Gültigkeitsbereiche definierten Namensauflösung für Konzepte der Basissprache. Eine Erweiterung, die das Basiskonzept für einen lokalen Gültigkeitsbereich erweitert (in DBL durch die Metaklasse LocalScopeStatement definiert), kann das Metaobjektattribut für die Anweisungen wiederverwenden. Anweisungen, die Variablen im Kontext der Erweiterung einführen, sind damit auch in Basisanweisungen sichtbar.

Direktableitungen

Regeln der Form $R \rightarrow Y_1, \ldots, R \rightarrow Y_i \ldots R \rightarrow Y_n$ sind besondere Regeln, die als Direktableitungen bezeichnet werden. Gibt es mehrere solcher Regeln, die auf ihrer

linken Seite das gleiche Nichtterminalsymbol R besitzen, so stellen die Nichtterminalsymbole $Y_1, \ldots, Y_n$ auf den rechten Seiten Spezialisierungen von R dar. Dort wo ein R steht, darf eine der Zeichenfolgen, die von $Y_1, \ldots, Y_n$ spezifiert werden, vorkommen.

Im Metamodell werden Direktableitungen durch Generalisierungsbeziehungen abgebildet. Die Metaklassen $Y_1, \ldots, Y_n$ sind jeweils als Spezialisierungen der Metaklasse R definiert. Durch diese Abbildung sind polymorphe Attribute vom Typ R im Metamodell möglich, die jeweils Referenzen auf die Objekte der Spezialisierungen $Y_1, \ldots, Y_n$ enthalten können.

Die Regeln der Spezialisierungen können wiederum Nichtterminalsymbole mit Syntaxattributen enthalten, die zu Attributen der speziellen Metaklassen werden. Dabei können gleichartige Attribute, die sich in ihrem Namen, ihrem Typ und ihrer Semantik nicht unterscheiden, mehrfach in Spezialisierungen enthalten sein. Diese gleichartigen Attribute werden nur in der Metaklasse der Basis R erfasst. Das hat den Vorteil, dass bei Zugriff auf Syntaxattribute vom Typ R keine Unterscheidung für jede Spezialisierung und auch kein Typ-Cast erfolgen muss. Die Beschreibung der Zugriffe wird damit einfacher.

Dabei sind zwei Fälle zu unterscheiden:

1. Ein Nichtterminalsymbol mit einem Syntaxattribut kommt in jeder Spezialisierung vor. Dann wird das entsprechende Attribut ohne Anpassungen in der Metaklasse der Basis R erfasst.

2. Ein Nichtterminalsymbol mit einem Syntaxattribut kommt in mindestens zwei Spezialisierungen, aber nicht in allen vor. Dann wird die Kardinalität des Attributes auf höchstens einen Wert gesetzt, so dass die Angabe eines Wertes optional ist.

Ähnliche Ableitungsansätze

Scheidgen [60] beschreibt, wie aus einer EBNF-Grammatik ein MOF-Metamodell abgeleitet werden kann. Die Ableitung erfolgt in zwei Schritten. Zunächst wird ein primitives Metamodell automatisch abgeleitet. Danach wird das Metamodell mit einer Semantikabbildung um zusätzliche semantische Informationen angereichert, so dass ein verbessertes Metamodell, das sowohl abstrakte als auch konkrete Konzepte enthält, abgeleitet werden kann.

Bei Scheidgen werden Spezialisierungen zwischen Metaklassen nicht automatisch aus Grammatikregeln abgeleitet. Stattdessen müssen in einem manuellen Prozess, als Teil der Beschreibung der Semantikabbildung, Beziehungen zwischen abstrakten und konkreten Metaklassen hinzugefügt werden.

Des Weiteren wird bei Scheidgen nicht zwischen attributierten und nicht-attributierten Symbolen unterschieden. Bei Symbolen auf den rechten Seiten einer Regel wird stets eine Assoziation zwischen Metaklassen erzeugt. Das Metamodell und die Objektgraphen werden damit wesentlich komplexer. Durch eine Unterscheidung, wie sie in ESL erfolgt, können Regeln definiert werden, bei denen nicht für jedes Nichtterminal auf einer rechten Seite ein neues Attribut in der Metaklasse eingeführt wird.

5.3.5 Implementierung

Der Ansatz DMX lässt sich mit dem Metamodellierungs-Framework EMF und dem Editor-Framework TEF implementieren. EMF bietet mit Ecore eine Metamodellsprache, die die Metakonzepte von MML enthält (siehe Abschnitt 5.3.1) und TEF erlaubt mit der Grammatiksprache TSL die Beschreibung einer Metamodellgrammatik unter Verwendung von MMGL-Metakonzepten (siehe Abschnitt 5.3.1). Damit sind beide Technologien für die Implementierung von DMX geeignet. Eine Herausforderung stellt jedoch die dynamische Änderung eines Ecore-Metamodells und einer TSL-Grammatik dar. Dazu wurde der durch das TEF-Framework bereitgestellte Modelleditor entsprechend angepasst. Die nachfolgenden Abschnitte beschreiben die dabei aufgetretenen Probleme. Eine prototypische Umsetzung, die die Implementierbarkeit der Syntaxerweiterung zeigt, ist mit Eclipse erfolgt und auf Github[2] verfügbar.

Anpassungen am Modelleditor TEF

Ein TEF-Editor wird mit einer TSL-Grammatik gestartet und bietet dann verschiedene Funktionen wie z.B. eine Syntaxhervorhebung und eine Autovervollständigung für die beschriebene Sprache an. Die TSL-Grammatik kann dabei während der Ausführung des TEF-Editors nicht verändert werden. Für das Framework DMX wurde TEF so angepasst, dass die TSL-Grammatik zur Laufzeit eines TEF-Editors austauschbar ist. Dieses angepasste TEF bildet die Grundlage für die Syntaxerweiterung, die in Java implementiert ist. Abb. 5.14 zeigt die Verwendung der Forever-Erweiterung im adaptiven Modelleditor.

forever-test.dbx

```
1 #import "forever"
2
3 module stdx_forevr_test;
4
5 active class A {
6     actions {
7         forever {
8             print time + "\n";
9             advance 1;
10        }
11    }
12 }
13
14 void main() {
15    A a = new A();
16    activate a;
17    advance 5;
18 }
```

Abbildung 5.14: Verwendung der Forever-Erweiterung im adaptiven Modelleditor.

Die Syntaxanalyse erfolgt bei TEF mit dem Parser RunCC [27], der einen LALR[3]-Parser und eine kontextfreie Grammatik zur Laufzeit interpretieren kann. Im Gegensatz zu einem Parser-Generator wie AntLR [53], bei dem der Quellcode des Parsers

[2] http://github.com/ablunk/dmx
[3] Look-Ahead, Left to right, Rightmost derivation

zunächst generiert werden muss, stellt RunCC einen solchen Parser durch Interpretation einer Grammatik bereit. Damit ist es möglich, die Grammatik zur Programmlaufzeit zu verändern.

Probleme der eingesetzten Parsing-Technik

Die in ESL beschriebene Ergänzung der kontextfreien Grammatik der Basissprache dient als Eingabe für denselben LALR-Parser, der auch für die Basissprache verwendet wird. Durch das Hinzufügen von Grammatikregeln kann als Ergebnis eine mehrdeutige Grammatik entstehen. Für die eingesetzte LALR-Technik kann dies ein Problem darstellen, da mit nur einem Look-Ahead-Symbol eindeutig entschieden werden muss, ob eine Shift- oder eine Reduce-Aktion auszuführen ist. Wenn dies nicht mehr eindeutig möglich ist, treten sogenannte Shift-Reduce- sowie Reduce-Reduce-Konflikte auf. Bei einem Shift-Reduce-Konflikt kann ein weiteres Symbol der Eingabe gelesen oder eine Reduktion vorgenommen werden. Bei einem Reduce-Reduce-Konflikt gibt es mehrere mögliche Regeln, die für eine Reduktion angewendet werden können. Letztlich können durch die Implementierung somit nur Spracherweiterungen vorgenommen werden, die zur Klasse der deterministisch kontextfreien Sprachen gehören.

Die Konflikte werden dem DSL-Entwickler mitgeteilt, indem die entsprechenden Meldungen des Parsers RunCC ausgegeben werden. Diese Ausgabe ist ohne Wissen über den LALR-Parsing-Algorithmus nur schwer zu verstehen. Der Entwickler muss Anpassungen, z.B. durch das Hinzufügen von Schlüsselwörtern vornehmen, so dass die Grammatik wieder eindeutig ist.

Das Auftreten dieser Konflikte ist kein grundlegendes Problem des vorgestellten Ansatzes. Aus praktischen Erwägungen wurde das Framework TEF, das einen LALR-Parser verwendet, für die vorliegende Arbeit verwendet und angepasst. Der LALR-Parser kann jedoch durch einen GLR-Parser (Generalized LR) [45] ausgetauscht werden. Dieser setzt einen Algorithmus ein, bei dem im Falle von Mehrdeutigkeiten mehrere parallele Parse-Bäume aufgebaut und verfolgt werden bis eindeutig einer dieser Bäume ausgewählt werden kann. Ein GLR-Parser unterstützt damit die Klasse der nicht-deterministisch kontextfreien Grammatiken.

Probleme des eingesetzten Metamodellierungs-Frameworks EMF

EMF bietet eine generische Java-basierte Schnittstelle für die Programmierung mit Metamodellen und deren Instanzen, sowie eine metamodellspezifische Schnittstelle, die einen typsicheren Zugriff auf ein spezifisches Metamodell erlaubt und durch die automatische Generierung von Interfaces und Klassen in Java erzeugt wird. Ein Java-Objekt repräsentiert eine Instanz einer Metaklasse und wird von Sprachwerkzeugen, die mit EMF implementiert sind, verwendet. Die Generierung und Verwendung von Java-Code zur Programmlaufzeit stellt jedoch ein schwieriges Problem dar, das vermieden wird, in dem die bereits vorhandene Java-Klasse für die Metaklasse des erweiterten Basiskonzeptes eingesetzt wird, um ein Java-Objekt zu erzeugen, das die Instanzen der Metaklasse der Erweiterung repräsentiert. Dazu wird die Metaklasse der Erweiterung über die generische EMF-Schnittstelle als Metaklasse des erzeugten Java-Objekts eingesetzt. Auf diese Weise lassen sich Werte für Attribute der Metaklasse der Erweiterung

über generische get- und set-Methoden auch für ein Java-Objekt setzen, das keine entsprechenden Java-Objektattribute besitzt.

5.4 Konzeptreduktion

Die Konzeptreduktion führt eine Erweiterung auf Basiskonzepte zurück. Dieses Vorgehen hat mehrere Vorteile. Der Transcompiler, der die Basissprache in eine bereits bestehende Programmiersprache übersetzt, kann für die Ausführung von Erweiterungen wiederverwendet werden. Wird eine Programmiersprache mit einer Semantikbeschreibung gewählt, die besonders laufzeiteffizient ausführbare Programme erlaubt, so kann diese Laufzeiteffizienz auf Erweiterungen übertragen werden. Dazu wird die Semantik einer Erweiterung als eine Abbildung auf ein Teilprogramm der Basissprache definiert. Die Konzeptreduktion ersetzt dann eine Erweiterungsinstanz in einem Basissprachenprogramm durch dieses Teilprogramm.

Die Zurückführung auf Basiskonzepte hat auch einen Nachteil. Es lassen sich nur Erweiterungen definieren, deren Semantik mit den Konzepten der Basissprache ausgedrückt werden können. Sprachkonzepte, deren Semantik auf diese Weise nicht definierbar ist, können nicht formuliert werden. Ein Beispiel ist ein Konzept, das eine Sichtbarkeit für Objektattribute einführt. Syntaktisch ist ein derartiges Konzept als Erweiterung beschreibbar. Die Semantik lässt sich jedoch nicht definieren. Dazu wäre es notwendig die Namensauflösung der Basissprache anzupassen, so dass in einem bestimmten Kontext, ein Zugriff auf ein geschütztes Objektattribut nicht mehr möglich ist.

Ein Konzept, dessen Semantik nicht durch eine Reduktion definiert werden kann, lässt sich nur durch eine Anpassung der Basissprache einführen. Dazu muss der Transcompiler für die Basissprache entsprechend angepasst werden. An dieser Stelle sind weitere Verbesserungen der Laufzeiteffizienz möglich, sobald sich ein Erweiterungskonzept etabliert hat. Diese Anpassung erfordert jedoch auch ein umfangreiches Wissen über die Definition der Basissprache, sowie der Sprache, in der die Semantik der Basissprache beschrieben ist. Die Abbildung auf die Basissprache, als Teil der Konzeptreduktion, kann dagegen in der Basissprache selbst erfolgen. Damit besitzt der Ansatz der Konzeptreduktion Vorteile bei der effizienten Entwicklung einer Sprache, da sich ein neues Sprachkonzept in der Basissprache selbst hinzufügen lässt. Der Ansatz erlaubt damit eine flexible Erweiterung einer Basissprache, die die laufzeiteffizienten Vorteile der Basissprache beibehält.

5.4.1 Analyse des abstrakten Syntaxbaums

Die Semantik einer Erweiterungsinstanz ist ein Teilprogramm der Basissprache, bezeichnet als Basisderivat, das anstelle der Erweiterungsinstanz eingesetzt wird und diese ersetzt. Das Basisprogramm muss zusammen mit dem Basisderivat ein syntaktisch korrektes Programm darstellen. Es besitzt eine Ausführungssemantik, die als Übersetzung in eine Zielsprache für die Basissprache bereits festgelegt ist.

Die Definition der Semantik einer Erweiterungsinstanz erfolgt durch Anweisungen in der Basissprache, die die Objektstruktur der Erweiterungsinstanz analysieren und eine Abbildung auf einen Teil des Basisderivats festlegen. Durch die Ausführung

der Anweisungen entsteht schrittweise das vollständige Basisderivat. Die dabei zu analysierende Objektstruktur der Erweiterungsinstanz ist durch die Syntaxattribute der ESL-Syntaxdefinition der Erweiterung festgelegt. Die Erweiterungsinstanz entspricht dabei einer Instanz des erweiterten Metamodells der Basissprache, das durch die ESL-Syntaxdefinition beschrieben ist.

Das Basisderivat wird mit einer speziellen Abbildungsanweisung erzeugt, die nur innerhalb von Semantikdefinitionen verwendet werden kann. Damit steht für die Semantikdefinition eine spezielle Teilsprache *Extension Reduction Language* (ERL) bereit, die die Sprache OBL zusammen mit einer Abbildungsanweisung umfasst.

Ausgehend von den Syntaxattributen, die in der Startregel definiert sind, kann durch Objektnavigation zu den Syntaxattributen der anderen Regeln navigiert werden. Für Syntaxsymbole, die auf ein Interface verweisen, und damit eine Metaklasse der Basissprache repräsentieren, erfolgt die Navigation durch den Aufruf entsprechender Get-Methoden. Dazu wird das Metamodell der Basissprache auf Interfaces und Methoden in der Basissprache abgebildet.

Abbildung des Metamodells der Basissprache auf die Basissprache

Die Sprache MML und die Basissprache besitzen viele ähnliche Konzepte, für die sich eine Abbildung einfach festlegen lässt. Eine Metaklasse wird auf ein Interface abgebildet. Die Basisklassen einer Metaklasse werden auf Basisinterfaces abgebildet. Metaobjektattribute werden zu get-Methoden im entsprechenden Interface. Es werden keine set-Methoden eingeführt, da die Metamodellinstanz im Rahmen einer Erweiterung lediglich analysiert, jedoch nicht verändert werden darf.

Die Wertesemantik für Metaobjektattribute kann jedoch in MML und in der Basissprache unterschiedlich sein. In MML ist es möglich eine Kompositionssemantik anzugeben, während in der Basissprache nur eine Referenzsemantik unterstützt wird. Ein Metaobjektattribut kann jedoch, unabhängig von seiner Wertesemantik, immer auf eine get-Methode abgebildet werden, da der Zugriff auf das Metamodell der Basissprache nur lesend erfolgt. Es muss lediglich sichergestellt werden, dass die Navigation über eine Metamodellinstanz mit den entsprechenden Interfaces möglich ist. Für jedes Metaobjektattribut einer Metaklasse wird also eine get-Methode im entsprechenden Interface hinzugefügt.

Falls das Metaobjektattribut eine Kardinalität als Liste mit beliebig vielen Werten festlegt, so wird als Rückgabetyp für die get-Methode ein Interface verwendet, das übliche Operationen für den Zugriff auf die Elemente einer Liste bereitstellt. Falls die Kardinalität höchstens einen Wert erlaubt, so wird als Rückgabetyp ein Interface oder ein Elementartyp verwendet. Listing 5.11 zeigt als Beispiel die Abbildung des MML-Metamodells für eine DBL-Funktion (siehe Abb. 5.6) auf die Basissprache. Als Basistyp wird hier EObject aus dem Ecore-Metamodell verwendet, da EMF als Implementierungstechnologie eingesetzt wird und die durch EMF bereitgestellte Modellverwaltung auf diese Weise wiederverwendet werden kann.

```
1 interface Function extends NamedElement, TypedElement, LocalScope {
2   bindings { "java" -> "hub.sam.dbl.Function" }
3
4   EList getParameters();
5   boolean isStatic();
```

```
6   boolean isAbstract();
7 }
8
9 interface NamedElement extends EObject {
10  bindings { "java" -> "hub.sam.dbl.NamedElement" }
11
12  string getName();
13 }
```

Listing 5.11: Beispiel für die Abbildung des MML-Metamodells für eine DBL-Funktion auf die Basissprache.

Wiederverwendung der Metaobjekt-Schnittstelle von EMF

EMF kann durch das Binding-Konzept von DBL wiederverwendet werden. Dazu wird das Ecore-Metamodell ebenfalls nach DBL abgebildet. Die Metaklasse EObject wird in das DBL-Interface in Listing 5.12 überführt. Damit kann ein DBL-Programm jede beliebige Instanz eines Ecore-Metamodells analysieren. Listing 5.13 zeigt eine DBL-Methode getContainer, die im angegebenen Objekt der Klasse EObject nach Eltern-Objekten sucht und das erste Eltern-Objekt zurückgibt, dessen Klassen dem angegebenen Namen entspricht. Mit dieser Methode kann ein Objekt gefunden werden, das ein anderes Objekt transitiv enthält.

```
1 interface EObject {
2   bindings {
3     "java" -> "org.eclipse.emf.ecore.EObject"
4   }
5   Resource eResource();
6   EClass eClass();
7   EObject eContainer();
8   EStructuralFeature eContainingFeature();
9   EReference eContainmentFeature();
10  EList eContents();
11  EList eAllContents();
12  EList eCrossReferences();
13  Object eGet(EStructuralFeature feature);
14  Object eGet(EStructuralFeature feature, boolean resolve);
15  void eSet(EStructuralFeature feature, Object newValue);
16  boolean eIsSet(EStructuralFeature feature);
17  void eUnset(EStructuralFeature feature);
18 }
```

Listing 5.12: Interface EObject mit Binding für EObject in Java.

5.4.2 Definition der Abbildung auf das Basisderivat

Das Basisderivat entsteht durch die Ausführung einer Semantikdefinition. Im Allgemeinen kann die Zurückführung auf das Basisderivat aus mehreren Konzeptreduktionen bestehen, da die Abbildung zunächst auf ein Derivat erfolgen kann, das nicht vollständig aus Basiselementen besteht und Erweiterungen enthält. Falls Erweiterungen enthalten sind, so werden diese durch Ausführung der entsprechenden Semantikdefinitionen ebenfalls auf ihr Basisderivat zurückgeführt. Nach der vollständigen Zu-

```
1  EObject getContainer(string parentClassName, EObject eObject) {
2    if (eObject != null) {
3      String eClassName = new String(eObject.eClass().getName());
4      if (eClassName.equals(parentClassName)) {
5        return eObject;
6      } else {
7        return getContainer(parentClassName, eObject.eContainer());
8      }
9    } else {
10     return null;
11   }
12 }
```

Listing 5.13: Zugriff auf das Eltern-Objekt eines Metaobjektes unter Verwendung des EMF-Bindings.

rückführung aller Erweiterungen auf ihr Basisderivat, entsteht ein Programm in der Basissprache, das keine Erweiterungen mehr enthält und mit einem Transcompiler für die Basissprache in ein ausführbares Programm übersetzt werden kann.

Für die Definition der Abbildung gibt es verschiedene Alternativen. Die Abbildung kann auf die konkrete Syntax, auf die abstrakte Syntax oder auf eine Kombination aus konkreter und abstrakter Syntax erfolgen. Die nachfolgenden Abschnitte beschreiben die Alternativen und gehen auf ihre Vor- und Nachteile ein.

Abbildung durch Angabe des Basisderivates in konkreter Syntax

Da die konkrete Syntax der Basissprache aus Text besteht, kann die Abbildung auf die konkrete Syntax durch die Angabe einer Zeichenkette erfolgen. Dabei wird das Basisderivat schrittweise durch Analyse der Werte von Syntaxattributen mit Operationen zur Konkatenierung von Zeichenketten aufgebaut.

Eine besondere Bedeutung haben dabei Syntaxattribute, die auf erweiterbare Sprachkonzepte verweisen. Verwendet man die konkrete Syntax dieser Elemente, so lässt sich auf einfache Weise ein Teil der Zeichenkette für das Basisderivat zusammenstellen. Die Voraussetzung ist, dass die konkrete Syntax jedes Basiselementes als Teil der abstrakten Syntax beim Parse-Vorgang gespeichert wird.

Da das Basisderivat selbst Erweiterungen enthalten kann, muss das Basisprogramm nach der Ersetzung erneut den Parse-Vorgang durchlaufen, so dass wieder eine Instanz des erweiterten Metamodells der Basissprache bereitsteht. Danach kann geprüft werden, ob noch Erweiterungsinstanzen vorhanden sind und die entsprechenden Abbildungen können vorgenommen werden. Dieser Vorgang wiederholt sich bis alle Erweiterungsinstanzen ersetzt sind.

Der Vorteil dieser Abbildung ist die Angabe des Basisderivats in der Notation der Basissprache, die dem Entwickler bereits vertraut ist und außerdem eine prägnante Darstellung von komplexen Objektstrukturen der abstrakten Syntax erlaubt. Des Weiteren ist auch eine prägnante Angabe anderer Erweiterungen in ihrer konkreten Syntax möglich, so dass eine Erweiterung zunächst auf andere Erweiterungen abgebildet werden kann.

Die Erweiterungsinstanzen bilden im Basisprogramm einen Baum. Eine Erweiterungsinstanz kann andere Erweiterungsinstanzen enthalten. Dabei müssen die Erweiterungsinstanzen an den Blättern des Baumes zuerst ersetzt werden. Je tiefer der Baum ist, desto häufiger ist eine erneute Übersetzung nach der Ersetzung notwendig. Damit verlängert sich die Übersetzungsdauer bis ein Basisprogramm mit Erweiterungen ausgeführt werden kann.

Einen weiteren Nachteil stellt das Fehler einer syntaktischen Prüfung bei der Angabe von Basiselementen in Zeichenketten dar. Da es sich hier lediglich um Fragmente eines Basisprogramms handelt, die erst durch die Ausführung der Abbildung ein syntaktisch korrektes Teilfragment bilden, ist eine Syntaxprüfung während der Eingabe nicht möglich. Dies kann leicht zu fehlerhaften Abbildungen führen, die erst bei der Ausführung erkannt werden.

Listing 5.14 zeigt ein Beispiel für eine Semantikdefinition. Dabei wird die Semantik der forever-Anweisung mit der expand-Anweisung als Abbildung auf eine while-Schleife festgelegt, die endlos lange ausgeführt wird. Die Abbildung erfolgt zunächst auf das Teilderivat while (true), das mit der konkreten Syntax für das Syntaxattribut body konkateniert wird. Im Ergebnis enthält das Basisderivat für die forever-Erweiterung eine while-Schleife, die aus den Anweisungen der forever-Anweisung besteht.

```
1  #import "../dbl"
2
3  module stdx_forever;
4
5  extension Forever extends dbl SimpleStatement {
6    start Forever;
7    Forever -> "forever" body:LocalScopeStatement;
8  }
9
10 semantics for Forever {
11   expand "while (true) " body;
12 }
```

Listing 5.14: Beispiel für die Erweiterungsdefinition einer Forever-Anweisung.

Die Sprache SLX [34] verfolgt einen ähnlichen Ansatz bei der Semantikbeschreibung. Hier wird mit einer expand-Anweisung die Zeichenkette für das Basisderivat schrittweise zusammengesetzt. Die Verwendung der konkreten Syntax eines Basiselementes ist dabei nicht möglich, da die Syntaxdefinition von SLX eine solche Angabe nicht erlaubt.

Als weitere verwandte Arbeit ist die Sprache Xtend [37] zu nennen. Xtend unterstützt besondere Zeichenketten, sogenannte Rich-Strings, die Anweisungen für bedingtes oder wiederholtes Einfügen von Zeichenketten enthalten können. Damit lässt sich ebenfalls ein Zielprogramm in seiner konkreten Syntax schrittweise zusammenbauen.

Abbildung durch Angabe des Basisderivates in abstrakter Syntax

Die abstrakte Syntax kann mit Anweisungen erzeugt werden, die Instanzen von Metaklassen der Basissprache erzeugen und diese miteinander in Beziehung setzen und mit

Werten befüllen. Dieses Vorgehen setzt voraus, dass die Metaklassen auf Klassen in der Basissprache abgebildet werden, von denen Objekte erzeugt werden können. Zusätzlich müssen Methoden für das Ändern von Werten vorhanden sein. Die Abbildung, die in Abschnitt 5.4.1 beschrieben ist, müsste dazu entsprechend erweitert werden.

Die Umsetzung kann ähnlich zum Metamodellierungs-Framework EMF für Java erfolgen. Für jede Metaklasse gibt es ein Interface und eine Klasse für die Instanzierung. Das Interface definiert get und set-Methoden zum Zugriff und zum Ändern von Werten. Die Klasse enthält Implementierungen für die Methoden entsprechend der Semantik der Metaobjektattribute. Zusätzlich gibt es eine Fabrik-Klasse mit einem Objekt, das für die Erzeugung von Instanzen verwendet wird.

Nach der Ausführung der Semantikdefinition entsteht ein Objekt in der Basissprache, das aus weiteren Objekten zusammengesetzt ist. Dieses Objekt muss auf entsprechende Instanzen der Metaklassen der Basissprache abgebildet werden. Danach kann die Erweiterungsinstanz durch die erzeugte Instanz einer Basismetaklasse ersetzt und die resultierende Metamodellinstanz für die weiteren Übersetzungsvorgänge verwendet werden.

Die Formulierung von Anweisungen zur Erzeugung der abstrakten Syntax ist komplexer als die Erzeugung einer konkreten Syntax, da eine kurze textuelle Darstellung durch eine umfangreiche Objektstruktur repräsentiert werden kann. Ein Beispiel stellen Ausdrücke dar. Jeder Teilausdruck ist ein Objekt, das weitere Teilausdrücke enthalten kann. Jedes dieser Objekte muss mit einer Anweisung erzeugt und mit einem Wert befüllt werden. Damit ist die Definition der Abbildung auf eine abstrakte Syntax nicht prägnant möglich und im Vergleich zur Angabe einer konkreten Syntax besonders umfangreich.

Der Sprachentwickler muss sich außerdem besonders gut im Metamodell der Basissprachen auskennen, um die Erzegung der entsprechenden Objektstrukturen zu beschreiben. Der Vorteil ist, dass dem Sprachentwickler weniger Fehler bei der Beschreibung der Abbildung unterlaufen können, da die erzeugbaren Objektstrukturen durch die entsprechenden Klassen klar vorgegeben sind.

Die Beschreibung einer Abbildung auf eine bestehende Erweiterung ist jedoch nicht möglich, da für jede Erweiterung das Metamodell der Basissprache dynamisch angepasst wird und keine entsprechenden Klassen in der Basissprache für die Erzeugung von Objekten vorhanden sind. Die Verwendung einer abstrakten Syntax bietet somit viele Nachteile.

Abbildung durch Angabe des Basisderivates mit Codeschablonen

Das Basisderivat besteht, wie jedes Basiskonzept, im Allgemeinen aus festen und variablen Syntaxsymbolen. Feste Symbole sind Schlüsselwörter oder Zeichen, die als Teil der konkreten Syntax des Basisderivates immer an einer bestimmten Stelle angegeben werden müssen. Dagegen sind für variable Teile verschiedene Ausprägungen möglich. Ein Beispiel ist das Basiskonzept Klasse, das immer mit dem Schlüsselwort `class` beginnt. Danach folgt ein variabler Teil in Form eines beliebigen Bezeichners, gefolgt von einem Paar aus geschweiften Klammern {}, zwischen denen Attribute und Methoden enthalten sein können, die selbst wieder aus festen und variablen Syntaxsymbolen bestehen.

Eine Codeschablone ist ein Ausdruck, mit dem ein Basiskonzept in konkreter Syntax angegeben wird. Der Typ des Ausdrucks ist die Metaklasse des Basiskonzeptes. Der Ausdruck kann also an eine Variable zugewiesen werden und durch Objektnavigation ausgewertet werden. In einer Codeschablone kann ein variabler Teil durch einen Ausdruck angegeben werden, der ein Basiskonzept enthält oder auf dieses verweist. Dabei ist die Angabe eines Syntaxattributes einer ESL-Syntaxdefinition, eines Objektes einer Basismetaklasse oder einer Codeschablone möglich.

Für jedes Basiskonzept lässt sich ein passender Ausdruck für eine Codeschablone aus den Grammatikregeln des Basiskonzeptes ableiten. Durch eine automatische Erweiterung der Basissprache ist es möglich jedes Basiskonzept in einer Semantikdefinition als Codeschablone anzugeben. Die Codeschablone bietet den Vorteil einer Syntaxprüfung durch den Modelleditor für das Basiskonzept, da die konkrete Syntax nur entsprechend der konkreten Syntax des Basiskonzeptes angegeben werden kann.

Der Ausdruck für den variablen Teil beschreibt eine Syntaxinjektion. Diese verweist auf ein Objekt einer Metaklasse, das im entsprechenden Attribut des Objektes der Codeschablone injeziert wird. Wird das entsprechende Attribut der Codeschablone angegeben, so ist kein erneutes Parsen der Codeschablone notwendig. Ohne eine explizite Angabe des Attributes kann nur durch eine erneute Auswertung der Grammatikregeln der Codeschablone entschieden werden, an welches Attribut die Zuweisung erfolgen muss. Durch die Angabe des Attributes kann außerdem statisch geprüft werden, ob ein Objekt in einer passenden Metaklasse angegeben ist.

Abb. 5.15 zeigt eine mögliche Definition für Codeschablonen im Metamodell und in der Grammatik der Basissprache. Damit eine Codeschablone ein beliebiges Metaobjekt enthalten kann, wird eine Wurzelklasse ModelElement eingeführt, von der alle Basismetaklassen erben. Eine Syntaxinjektion enthält einen Ausdruck, der auf ein Metaobjekt verweist. Unterhalb des Metamodells werden Ausschnitte der Grammatik gezeigt. Eine Codeschablone beginnt mit einem Ausfrufezeichen. Danach folgt eine Syntax, die spezifisch für das jeweilige Basiskonzept ist und mit einem spezifischen Schlüsselwort beginnt, um Mehrdeutigkeiten durch die Grammatikregeln der jeweiligen Basiskonzepte zu vermeiden. Die Grammatikregel für ein Basiskonzept wird für variable Teile so ergänzt, dass an diesen Stellen auch eine Syntaxinjektion stehen kann. Im Beispiel definiert die Regel InjectableName für den Bezeichner einer Klasse als Alternative eine Syntaxinjektion. Somit kann anstelle eines Klassennamens auch ein Verweis auf eine Variable vom Typ String stehen.

Die Verwendung einer Codeschablone wird in Abb. 5.16 am Beispiel der Semantikdefinition der Forever-Anweisung dargestellt. Die Codeschablone beschreibt eine While-Schleife, deren Anweisungen im Metaobjektattribut body des WhileStatement-Objektes durch das referenzierte Metaobjekt im Syntaxattribut foreverBody injeziert werden.

Es gibt bereits Arbeiten, die sich mit der automatischen Ableitung von Codeschablonen aus der Grammatik einer Sprache beschäftigen. Wachsmuth [80] beschreibt einen Algorithmus zur Ableitung einer Codeschablonensprache für eine festgelegte Zielsprache, die die syntaktische Korrektheit von Zielprogrammen garantiert, die mit den Codeschablonen formuliert werden. Arnoldus [3] entwickelt in seiner Arbeit mit *Syntax Safe Templates* einen ähnlichen Ansatz und beschreibt zusätzlich eine prototypische Implementierung. Weber [82] beweist in seiner Arbeit die Äquivalenz in

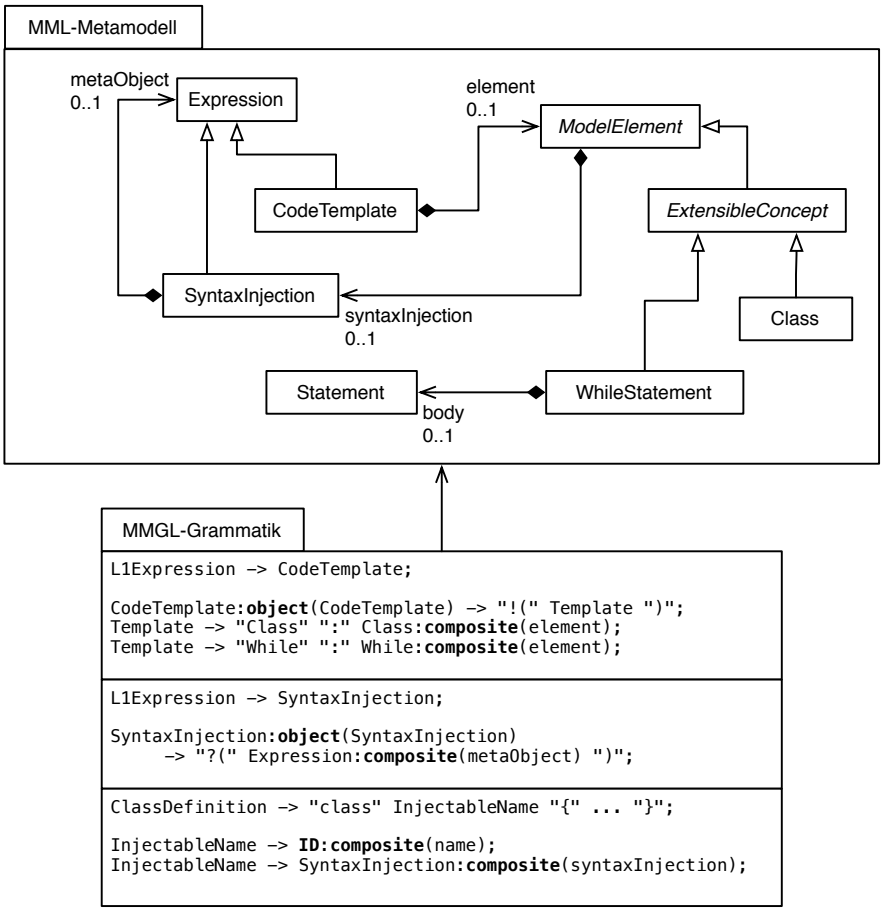

Abbildung 5.15: Definition von Codeschablonen im Metamodell und in der Grammatik.

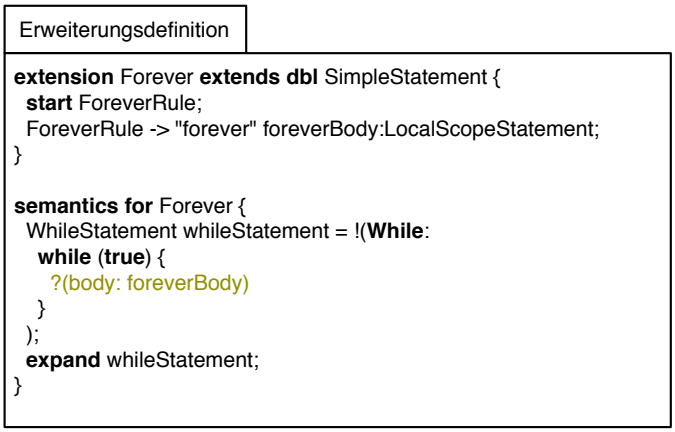

Abbildung 5.16: Beispiel für die Verwendung einer Codeschablone zur Definition der Semantik für die Forever-Anweisung.

der Ausdruckskraft zwischen Codeschablonen und kontextfreien Grammatiken mit einem formalen Modell. Diese Arbeiten können als Ausgangspunkt für Untersuchungen zur Integration von Codeschablonen in den Ansatz DMX dienen.

Abbildung in einem anderen Kontext

Die Sprachelemente Klasse, Funktion, Variable und Anweisung können in verschiedenen Container-Sprachelementen auftreten und verändern die Semantik des Programms nicht, wenn für deren Benennung keine bereits existierenden Bezeichner verwendet werden. Das Hinzufügen dieser Elemente ist sinnvoll, wenn die Semantik einer Erweiterungsinstanz auf Sprachelemente angewiesen ist, die in ihrem Kontext nicht hinzugefügt werden können. Eine Statement-Erweiterung, die eine noch nicht vorhandene Klasse verwendet, hat so die Möglichkeit diese Klasse im umgebenden Modul hinzufügen. Da ein solches Hinzufügen keine bestehenden Klassen verändert, ist diese Änderung im Rahmen von Erweiterungen zulässig.

Die Angabe eines Kontextes ist optional in einer expand-Anweisung erlaubt. Der Kontext besteht aus einer Angabe des Typs des erzeugten Elementes und aus einem Verweis auf ein passendes Metaobjekt des umgebenden Programms. Als Typ sind die Angaben class, function und variable erlaubt. Listing 5.15 zeigt mögliche Beispielaufrufe für eine kontextverändernde expand-Anweisung.

```
1 expand class "class A{}" in containerModule;
2 expand function "void m() {}" in containerClass;
3 expand variable "int v;" in containerClass;
```

Listing 5.15: Beispielaufrufe für eine kontextverändernde expand-Anweisung.

Ein Beispiel ist eine Statement-Erweiterung, die auf eine Anweisung abgebildet wird, die ein Objekt einer nicht vorhandenen und somit ebenfalls zu erzeugenden neuen Klasse erzeugt. Die Klasse muss im Kontext des Moduls erzeugt werden, das als Teil eines entsprechenden Elementes, z.B. einer Funktion, die Statement-Erweiterung enthält. Listing 5.16 zeigt die Erweiterung FX, die eine neue Klasse A im umgebenden Modul erzeugt und dann für die Erzeugung eines A-Objektes im Zielcode verwendet. Das umgebende Modul wird durch die Funktion getParentModule extrahiert.

```
1 extension FX extends dbl SimpleStatement {
2   start FX;
3   FX -> "fx" name:ID ";";
4 }
5
6 semantics for FX {
7   expand class "class A {}" in getParentModule(self);
8   expand "A " name " = new A();";
9 }
```

Listing 5.16: FX-Erweiterung als Beispiel für den Einsatz einer setExpand-Anweisung.

Eindeutige Bezeichner im Kontext einer Ersetzung

Bei Erweiterungen, die neue benannte Element erzeugen, kann es zu Namenskonflikten mit bereits existierenden Namen oder durch Erweiterungen eingeführte Namen

```
extension GX extends dbl SimpleStatement {
  start GX;
  GX -> "gx" ";";
}

semantics for GX {
  ID i;

  expand "int " i " = 0;";
}
```

Listing 5.17: GX-Erweiterung als Beispiel für den Einsatz einer ID-Anweisung.

```
void main() {
  gx;
  gx;
}
```

Listing 5.18: GX-Erweiterunginstanzen.

kommen. Benannte Elemente können z.B. Variablen, Funktionen und Klassen sein. Erweiterbare Sprachen [4] bieten als Hilfe das hygenische Einführen von Namen an. In ERL gibt es dazu eine spezielle ID-Anweisung, die einen neuen eindeutigen Namen einführt. Wird dieser Name in einer expand-Anweisung verwendet, so wird anstelle des Namens stets ein eindeutiger Name verwendet.

Ein Beispiel für die Verwendung der ID-Anweisung wird in den Listings 5.17 und 5.18 gezeigt. Jede GX-Erweiterungsinstanz erzeugt eine int-Variable mit einem eindeutigen Namen. Dieser eindeutige Name erlaubt die mehrfache Verwendung dieser Erweiterung im gleichen Kontext.

5.4.3 Implementierung

Die Konzeptreduktion erfolgt schrittweise durch die Ersetzung aller Erweiterungsinstanzen, indem für jede Erweiterungsinstanz die zugehörige Semantikdefinition ausgeführt wird. Für die Implementierung muss eine Semantikdefinition in ein ausführbares Programm übersetzt werden, das auf die Erweiterungsinstanz zugreifen und diese, entsprechend den Anweisungen der Semantikdefinition, analysieren kann. Da die Sprache OBL in der Sprache ERL enthalten ist und für OBL bereits ein Transcompiler nach Java existiert, kann der OBL-Java-Transcompiler für die Übersetzung von ERL nach Java wiederverwendet werden. Zusätzlich muss der Zugriff auf die Erweiterungsinstanz sowie die Übersetzung der Abbildungsanweisung und der ID-Anweisung realisiert werden.

Zugriff auf eine Erweiterungsinstanz

Für den Zugriff auf eine Erweiterungsinstanz ist der Einsatz eines Metamodellierungsframeworks sinnvoll. Da das Metamodell für DBL in Ecore definiert ist, wird EMF als Java-basiertes Metamodellierungsframework für den Zugriff auf Ecore-Metamodellin-

stanzen verwendet. Das Java-Programm erhält als Argument eine Erweiterungsinstanz und erzeugt ein Basisderivat, das die Erweiterungsinstanz ersetzt und an allen definierten Stellen im umgebenden Basisprogramm Sprachelemente hinzufügt.

Meta-Objekte werden in EMF durch Java-Objekte repräsentiert. Für jedes Java-Objekt muss es dabei eine entsprechende Java-Klasse geben, die einer Klasse im Metamodell entspricht. Die notwendigen Java-Klasse sowie weitere Hilfsklassen zur Erzeugung der Java-Repräsentationen von Meta-Objekten sind abhängig von einem konkreten Metamodell und werden für dieses in einem Code-Generierungsschritt erzeugt.

Für das DBL-Metamodell sind die entsprechenden Java-Klassen bereits vorhanden. Da das DBL-Metamodell jedoch dynamisch um neue Elemente erweitert wird, müssen entsprechend der neu hinzugefügten Metaklassen weitere Java-Klassen generiert sowie kompiliert werden. Dieser Schritt benötigt viel Zeit während der Auflösung und Übersetzung eines DBL-Programms mit Erweiterungen.

Neben dem typisierten Zugriff über die generierte Java-Schnittstelle, gibt es jedoch auch eine untypisierte allgemeine Schnittstelle zur Arbeit mit Meta-Objekten, die als EMF-Reflection bekannt ist und durch die Klasse EObject beschrieben ist. Diese Schnittstelle erlaubt es auf Eigenschaften und Werte zuzugreifen, ohne dass dazu eine spezielle Java-Schnittstelle generiert werden muss. Da der Zugriff untypisiert erfolgt, sind viele Type-Casts notwendig, die jedoch automatisch bei der Übersetzung einer Semantikbeschreibung nach Java eingefügt werden. Durch den Einsatz von EMF-Reflection wird die Dauer der Übersetzung verkürzt.

Schrittweise Reduktion

Die Reduktion erfolgt durch die Abarbeitung der folgenden Schritte.

1. Es werden alle Erweiterungsinstanzen gesucht, die selbst keine Erweiterungsinstanzen enthalten. Falls noch Erweiterungsinstanzen gefunden werden, so wird mit Schritt 2 fortgefahren. Sonst ist die Konzeptreduktion abgeschlossen.

2. Für jede Erweiterungsinstanz wird die passende Erweiterungsdefinition gesucht. Dazu wird die Metaklasse der Erweiterungsinstanz verwendet, da jede Erweiterungsdefinition nach einem festen Namensschema eine Metaklasse für Erweiterungsinstanzen dem Basismetamodell hinzufügt.

3. Für jede Erweiterungsdefinition wird die zugehörige Semantikdefinition mit dem ERL-Java-Transcompiler in ein Java-Programm übersetzt. Das Java-Programm erhält als Parameter das Basisprogramm als Metamodellinstanz und einen Verweis auf die zu übersetzende Erweiterungsinstanz im Basisprogramm.

4. Für jede Erweiterungsinstanz aus Schritt 2 wird das zugehörige Java-Programm der Erweiterungsdefinition ausgeführt und das Basisderivat als Ergebnis der Abbildung wird gespeichert.

5. Jede Erweiterungsinstanz wird durch das gespeicherte Basisderivat ersetzt. Der Ersetzungsvorgang ist abhängig von der Art der Abbildung, die unter Angabe der konkreten Syntax, der abstrakten Syntax oder mit Codeschablonen erfolgen kann. Im Rahmen der Implementierung im Framework DMX wurde die Abbildung auf die konkrete Sytax umgesetzt.

a) Konkrete Syntax: Das Basisprogramm wird in diesem Fall als Zeichenkette verarbeitet. Die Ersetzung setzt voraus, dass für jede Instanz einer Metaklasse ihre Position in der Zeichenkette bekannt ist. Diese Information kann beim Parsen des Basisprogramms und der Erzeugung der Metamodellinstanz bereits hinterlegt werden. Die Position und die Länge der Erweiterungsinstanz in der Zeichenkette für das Basisprogramm werden durch die Zeichenkette für das Basisderivat ersetzt. Des Weiteren werden Zeichenketten, die durch Abbildungsanweisungen mit Kontextänderung vorhanden sind, an den entsprechenden Positionen der referenzierten Metaobjekte hinzugefügt. Danach muss die resultierende Zeichenkette erneut mit dem Parser in eine Metamodellinstanz übersetzt werden, so dass die weiteren Schritte der Konzeptreduktion durchgeführt werden können.

b) Abstrakte Syntax und Codeschablonen: Die Metamodellinstanz, die das Basisprogramm repräsentiert, kann in diesen beiden Fall direkt verwendet werden. Das Basisderivat liegt bereits als Metaobjekt vor. Die Erweiterungsinstanz wird durch das Metaobjekt für das Basisderivat in der Metamodellinstanz ersetzt. Ein erneutes Parsen des Basisprogramm ist dabei nicht notwendig.

6. Nachdem alle Erweiterungsinstanzen, die selbst keine Erweiterungsinstanzen enthalten, ersetzt sind, wird wieder mit Schritt 1 begonnen. Dabei werden als nächstes Erweiterungsinstanzen verarbeitet, die Basisderivate enthalten sowie Basisderivate, die Erweiterungsinstanzen enthalten.

Die Konzeptreduktion ist unabhängig von der Zielsprache, die für die Simulation verwendet wird. Die Reduktion erfolgt zunächst vollständig in Java. Sobald das Basisprogramm keine Erweiterungsinstanzen mehr enthält, kann ein anderer Transcompiler für die Übersetzung des Basisprogramms eingesetzt werden, so dass ein besonders laufzeiteffizientes Programm für die Durchführung der Simulation verwendet werden kann.

5.5 Möglichkeiten der Erweiterbarkeit

Die Erweiterbarkeit einer Basissprache ist beim Ansatz DMX auf die Einführung von alternativen syntaktischen Ausprägungen beschränkt, deren Semantik als Abbildung auf bestehende Basiskonzepte ausgedrückt werden kann. Eine Erweiterung erlaubt das Hinzufügen von Sprachkonzepten, jedoch keine Änderung an bestehenden Sprachkonzepten.

Die Syntaxdefinition von DBL ist so vorbereitet, dass die Hauptsprachelemente erweiterbar sind. Die meisten Bestandteile dieser Sprachelemente sind jedoch nicht erweiterbar. Hauptsprachelemente sind Klassen, Interfaces, Variablen, Funktionen, Anweisungen, Ausdrücke und Erweiterungen. Die Bestandteile dieser Sprachelemente sind nur dann erweiterbar, wenn der Bestandteil auch ein Hauptsprachelement ist. Ein nicht erweiterbarer Bestandteil einer Klasse ist z.B. die Angabe von Basisklassen. Erweiterbar sind dagegen Variablen, Funktionen und Anweisungen, als Bestandteile einer Klassendefinition.

Erlaubt man Erweiterungen für Bestandteile, die keine Hauptsprachelemente sind, so können diese Veränderungen an der Semantik eines Hauptsprachelementes erforderlich machen. Eine Umsetzung würde jedoch Veränderungen am Hauptsprachelement notwendig machen. Damit würde keine Erweiterung mehr vorliegen, da die Semantik der Basissprache verändert wird.

Aus diesem Grund sind auch die Modifizierer für Variablen und Funktionen, wie z.B. *abstract*, nicht erweiterbar. Würden Erweiterungen für Modifizierer erlaubt sein, so müsste das Sprachelement, für das der Modifizierer definiert ist, verändert werden. Damit wäre es aber keine Erweiterung, sondern eine Veränderung. Beispiele für weitere Modifizierer sind eine Sichtbarkeitseigenschaft, die den Zugriff auf eine Variable aus einem anderen Kontext einschränkt. Die Umsetzung würde jedoch eine Veränderung der Namensauflösung und das Hinzufügen von statischen Semantikregeln erfordern. Erweiterungen, die Veränderungen an der Semantik der Basissprache erforderlich machen, sind also nicht erlaubt, da dies ebenfalls Veränderungen an den Sprachwerkzeugen nach sich ziehen würde.

Eine Ausnahme von der Regel ist für Sprachelemente im Kontext von Modulen, Klassen und Interfaces sinnvoll. Diese Sprachelemente enthalten Hauptsprachelemente, die erweiterbar sind. Ein Einführung einer Erweiterung für ein solches Hauptsprachelement führt dazu, dass eine Erweiterung an den gleichen Stellen, wie das Hauptsprachelement referenzierbar ist. Dies ist jedoch nicht immer sinnvoll. Deshalb können Erweiterungen im Kontext von Modulen, Klassen und Interfaces über spezielle Basiskonzepte, die keine konkreten Ausprägungen besitzen, eingeführt werden.

5.6 Möglichkeiten zur Wiederverwendung von Konzepten

DMX unterstützt verschiedene Möglichkeiten zur Wiederverwendung von Sprachkonzepten, mit dem Ziel den Aufwand zur Entwicklung von Erweiterungen zu reduzieren. Dabei können sowohl Elemente der Basissprache als auch Erweiterungen in Erweiterungen verwendet werden.

Die erste Form der Wiederverwendung sind *Erweiterungen, die Basiselemente enthalten oder auf diese verweisen*. Damit lassen sich z.B. Anweisungen als Erweiterungen definieren, die wiederum Anweisungen und Ausdrücke der Basissprache enthalten. Die ForEach-Anweisung ist eine Beispiel für eine derartige Erweiterung (siehe Listing 5.19).

```
1  extension ForEach extends dbl LocalScopeStatement {
2    start ForEach;
3    ForEach -> "foreach" "(" statements : list Variable "in" collection : Expression ")"
4      body : LocalScopeStatement;
5  }
6
7  semantics for ForEach {
8    Variable itemVariable = statements.get(0) as Variable;
9
10   ID it;
11   ID item;
12   expand "Iterator " it " = " collection ".iterator();";
13   expand "while (" it ".hasNext()) {";
14   expand " Object " item " = " it ".next();";
```

```
15   expand   itemVariable " = " item " as " itemVariable.getClassifierType() ";";
16   expand  body;
17   expand "}";
18 }
```

Listing 5.19: ForEach-Anweisung als Erweiterung.

Die zweite Möglichkeit zur Wiederverwendung sind Erweiterungen, deren *Semantik als eine Abbildung auf Basiselemente oder Erweiterungen* definiert ist. Auf diese Weise kann der bestehende Compiler für die Basissprache auch für Erweiterungen wiederverwendet werden, da Erweiterungen immer durch Basissprachelemente ersetzt werden.

Eine dritte Form der Wiederverwendung stellen *modulare Erweiterungen* dar. Eine Erweiterung A kann andere Erweiterungen B enthalten. Da jede Erweiterung selbst auch erweiterbar ist, können anstelle von B auch Erweiterungen angegeben werden, die B erweitern und damit an der Stelle von B verwendet werden können. Genauso kann eine Erweiterung A auch Verweise auf Erweiterungen B enthalten. Dann ist es möglich an der Stelle des Verweises auch Erweiterungen anzugeben, die B erweitern.

Die vierte Möglichkeit erlaubt die *Wiederverwendung der abstrakten Syntax* eines Basiselementes. Da Sprachwerkzeuge auf der Grundlage der abstrakten Syntax implementiert sind, können diese für die Basismerkmale ebenfalls wiederverwendet werden.

Ein Beispiel ist eine Erweiterung des Basiselementes LocalScopeStatement. Das Sprachwerkzeug zur Namensauflösung ist das *DBL Identification Scheme*, das als Teil des adaptiven Modelleditors implementiert ist. Es definiert Regeln für die Namensauflösung von Variablen in lokalen Gültigkeitsbereichen. Bei einer Erweiterung, die das Sprachelement LocalScopeStatement erweitert und die Einführung lokaler Variablen über den strukturellen Symbolverweis statements erlaubt, können die in der Erweiterung enthaltenen Anweisungen auf die eingeführten Variablen zugreifen. Dabei werden die Regeln zur Namensauflösung, die für jedes LocalScopeStatement definiert sind, benutzt. Ein Beispiel, das hiervon Gebrauch macht ist die ForEach-Erweiterung. Würde ForEach als einfache SimpleStatement-Erweiterung definiert werden, so könnten die als Teil ihres Körpers formulierten Anweisungen nicht auf die eingeführten Variablen zugreifen.

5.6.1 Arten von Erweiterungen

Die Erweiterungsmöglichkeiten der Basissprache lassen sich in folgende Klassen unterteilen:

1. Alternative,
2. Konstruktion und
3. Spezialisierung.

Eine *Alternative* fügt eine neue syntaktische Form für ein Basiskonzept hinzu. Durch die Syntaxerweiterung gibt es eine Grammatikregel, die eine Alternative für die Grammatikregel des Basiskonzeptes definiert sowie eine Metaklasse, die die Metaklasse des Basiskonzeptes spezialisiert. Damit kann die Erweiterung an allen Stellen, an denen auch das Basiskonzept stehen kann, verwendet werden. Außerdem kann die

Erweiterung auch anstelle des Basiskonzeptes referenziert werden. Ein Beispiel für eine Alternative ist die Forever-Anweisung.

Eine *Spezialisierung* ist eine besondere Alternative. Dabei wird ein Metaobjektattribut eines erweiterten Basiskonzeptes für ein Syntaxattribut in der Erweiterungsdefinition wiederverwendet. Auf diese Weise können Sprachwerkzeuge, die auf der Grundlage des Metamodells der Basissprache implementiert sind, die Erweiterung so wie das Basiskonzept behandeln. Ein Beispiel ist eine Datenklasse als spezielle Klasse (Definition siehe Listing 5.20). Bei einer Datenklasse wird der Konstruktor der Klasse aus den Objektattributen erzeugt. Der Modelleditor der Basissprache behandelt eine Datenklasse genauso wie eine Klasse. Damit sind Attribute einer Datenklasse genauso referenzierbar wie Attribute einer Klasse (siehe Listing 5.21).

```
#import "../../dbl"
#import "../../stdx"

module dataClass;

extension DataClass extends dbl Class {
  start DataClass;
  DataClass -> "data" name:ID "{" Attributes "}";
  Attributes -> ;
  Attributes -> attributes : list Variable ";" Attributes ;
}

semantics for DataClass {
  expand "class " name " {";
  foreach (Variable attribute in attributes) {
    expand attribute ";";
  }

  expand "new(" attributes ") {";
  foreach (Variable attribute in attributes) {
    expand "self." attribute.getName() "=" attribute.getName() ";";
  }
  expand "}";

  expand "}";
}
```

Listing 5.20: Erweiterungsdefinition einer Datenklasse als Erweiterung einer Klasse.

```
#import "dataclass"

module dataClassTest;

class A {
  int i;
  new (int i) {
    self.i = i;
  }
}

data D {
  int i;
}
```

```
15
16 void main() {
17   A a = new A(1);
18   D d = new D(1);
19   d.i = 1;
20 }
```

Listing 5.21: Zugriff auf Objektattribute einer Datenklasse.

Eine *Konstruktion* ist ebenfalls eine spezielle Alternative. Sie erfolgt im Kontext eines Basiselementes für ein Basiselement, das keine konkrete Ausprägung besitzt. Eine Konstruktion definiert einen abstrakten Ersetzungspunkt. Sie erlaubt damit das Einfügen vollkommen neuer Sprachelemente, die keine Alternative zu einem bestehenden Basiselement darstellen. Ein Basiskonzept erlaubt Erweiterungen als Konstruktionen wenn das Basiskonzept durch eine leere Metaklasse definiert ist und es eine zugehörige Grammatikregel gibt. In DBL sind Konstruktionen in einem Modul (als Erweiterung von ModuleContentExtension) und in einer Klasse (als Erweiterung von ClassContentExtension) möglich.

Ein Beispiel ist eine Zustandsmaschine, die im Kontext einer Klasse definiert wird (Definition siehe Listing 5.22 und Anwendung siehe Listing 5.23). Die Attribute und Methode der umgebenden Klasse können dabei in Anweisungen und Ausdrücken der Zustandsmaschine verwendet werden. Der Name eines Attributes bzw. einer Methode wird dabei auf der Grundlage der Namensauflösung für das Basiskonzept, in diesem Fall einer Klasse, aufgelöst.

```
1 extension StateMachine extends dbl ClassContentExtension {
2   start StateMachine;
3
4   StateMachine -> "stateMachine" "{" ManyRegularStates "}";
5   ...
6 }
```

Listing 5.22: Ausschnitt einer Erweiterungsdefinition einer Zustandsmaschine als Teil einer Klasse.

```
1 active class A {
2   int c = 0;
3
4   state machine {
5     initial state s {
6       after 1 do { c = c + 1; m(); } then s
7     }
8   }
9
10  void m() {}
11 }
```

Listing 5.23: Erweiterungsinstanz für eine Zustandsmaschine in einer Klasse.

Tabelle 5.1: Prioritätsklassen für Operatoren in DBL.

Priorität	Operator	Operationen	Grammatikregel
1	. ()	member access, procedure call	L1Expr
2	+ - !	unary plus, minus, negation	L2Expr
3	* / %	multiplicative	L3Expr
4	+ -	additive	L4Expr
5	< > <= >= instanceof	relational	L5Expr
6	== !=	equality	L6Expr
7	and	logical and	L7Expr
8	or	logical or	L8Expr

5.6.2 Grenzen

Keine neuen Prioritätsklassen für Ausdrücke

Eine Grenze für die Syntaxerweiterung stellt die Erweiterung von Ausdrücken dar. Die Auswertung eines Ausdrucks ist von der Priorität der beteiligten Operatoren abhängig. Damit der abstrakte Syntaxbaum eines Ausdrucks die Priorität abbildet, enthält eine kontextfreie Grammatik Regeln für die verschiedenen Prioritätsklassen. Eine Prioritätsklasse umfasst alle Operatoren mit der gleichen Priorität. Die Sprache DBL definiert 9 Prioritätsklassen (siehe Tabelle 5.1). Die Erweiterung von Ausdrücken ist dabei nur für vorhandene Prioritätsklassen möglich, da die Einführung einer neuen Prioritätsklasse eine Änderung der Grammatik erfordert.

Listing 5.24 zeigt einen Ausschnitt der Grammatik von DBL, in dem die Ausdrücke für logisches Oder und logisches Und definiert werden. Diese beiden Prioritätsklassen besitzen eine geringe Priorität. Ein Ausdruck wird zunächst auf die Prioritätsklasse für ein logisches oder (OrClassExpression) abgeleitet. Diese Prioritätsklasse umfasst nur den Ausdruck für ein logisches Oder (Or).

Die Einführung eines ternären If-Else-Ausdrucks ist durch Erweiterung nicht möglich. Dieser Ausdruck hat eine geringere Priorität als das logische Oder. Die Regel Expression müsste für die Erweiterung redefiniert werden und eine Ableitung auf eine Regel TernaryClassExpression für die neue Prioritätsklasse definieren. Erst TernaryClassExpression darf eine Ableitung auf die Regel für die Prioritätklasse des logischen Oder festlegen. Eine Anpassung der Grammatik ist mit der Syntaxerweiterung jedoch nicht möglich.

Keine Auflösung während der Modellierung

Alternative, Konstruktion und Spezialisierung sind Erweiterungsarten, die erst vom DBL-Compiler in Elemente der Basissprache übersetzt werden. Damit ist es möglich auf diese Erweiterungen an anderen Stellen eines DBL-Programms zu verweisen. Sprachwerkzeuge, die DBL-Programme vor ihrer Übersetzung verarbeiten, wie z.B. der textuelle Editor, besitzen keine Informationen über das Ergebnis der Abbildung. Sie können ihre Funktionalität also nur für die Basiselemente zur Verfügung stellen.

```
1  Expression −> OrClassExpression;
2  OrClassExpression −> Or;
3  OrClassExpression −> AndClassExpression;
4  AndClassExpression −> And;
5  AndClassExpression −> EqualityClassExpression;
6
7  Or:element(Or) −> OrClassExpression:composite(operand1)
8      "or" AndClassExpression:composite(operand2);
9  And:element(And) −> AndClassExpression:composite(operand1)
10     "and" EqualityClassExpression:composite(operand2);
```

Listing 5.24: Definition von Ausdrücken in der Grammatik von DBL.

Eine *Auflösung* ist eine spezielle Erweiterung, die bereits während des Editierens im Hintergrund auf Basiselemente abgebildet und durch diese ersetzt wird. Die Ersetzung erfolgt unsichtbar für den Benutzer. Dem Benutzer wird die Erweiterung mit der festgelegten Syntax präsentiert. Sprachwerkzeuge erhalten die Erweiterung jedoch in ihrer Repräsentation aus Basiselementen. Die Auflösung wird also unsichtbar für den Benutzer im DBL-Programm nach außen gestülpt. Diese Form der Erweiterung kann auch als *Inside-Out-Erweiterung* bezeichnet werden.

Erweiterungen, die nur auf Basis einer Auflösung sinnvoll definiert werden können, sind die aus C++ bekannten Templates und die aus Java bekannten Generics. Nur bei einer Auflösung, können andere Teile eines DBL-Programms auf Basis eines spezifisches Typs auf die Attribute und Methoden einer Klasse mit Typ-Parametern zugreifen.

Ein andere Erweiterung, die nur durch Auflösung verwendet werden kann, ist eine Erweiterung der Syntaxbeschreibungssprache ESL um die Konzepte von EBNF. Im Hintergrund des Editors läuft permanent ein Prozess, der die Syntax von Erweiterungen analysiert und die Grammatik und das Metamodell von DBL entsprechend erweitert. Dieser Prozess ist auf der Grundlage der Basiselemente definiert. Eine Erweiterung um ein Wiederholungs-Konstrukt aus EBNF kann dieser Prozess nicht sinnvoll verwerten, da die Semantik des neuen Konstrukts nicht Teil des Prozesses ist und in der Semantikbeschreibung der Erweiterung formuliert ist. Diese wird jedoch erst vom DBL-Compiler ausgewertet. Nur durch eine Auflösung, die eine Erweiterung noch im Editor in Basisselemente übersetzt, kann der Prozessor auch für Erweiterungen der Syntaxsprache verwendet werden.

5.6.3 Erweiterung von Erweiterungen

Eine Erweiterungsdefinition erfüllt die gleichen Voraussetzungen wie erweiterbare Sprachelemente (siehe 5.3.3) und ist damit ebenfalls erweiterbar. Die Syntaxerweiterung bildet eine Erweiterungsdefinition auf eine Metaklasse und eine Regel ab, die den gleichen Namen besitzen. Dabei ist die Regel nicht an eine Metaklasse gebunden und definiert eine Ableitung auf eine weitere Regel, die erst die Bindung an die Metaklasse festlegt (siehe 5.3.4). Damit ist Voraussetzung 1 erfüllt. Nach Voraussetzung 2 ist die eingeführte Metaklasse außerdem nicht abstrakt. Zur Instanziierung wird die Basismetaklasse derjenigen erweiterten Erweiterung verwendet, die ein Basiskonzept

```
extension A extends dbl Statement {
  start A;
  A -> "a" b:B;
}

extension B {
  start B;
  B -> ;
}

extension C extends extension B {
  start C;
  C -> "c";
}
```

Listing 5.25: Beispiel für eine Erweiterung einer Erweiterung.

erweitert. Dabei kann eine Erweiterung eine andere Erweiterung über mehrere andere Erweiterungen erweitern. Die Voraussetzung 3 ist erfüllt, da die Voraussetzung bereits für die erweiterte Erweiterung erfüllt sein muss. Gibt es für die erweiterte Erweiterung ein Elternkonzept, so ist dieses auch für eine Erweiterung dieser Erweiterung vorhanden. Damit sind alle Voraussetzungen für erweiterbare Sprachkonzepte erfüllt.

Neben der Erweiterung einer Erweiterung ist auch eine Komposition einer Erweiterung aus Erweiterungen möglich. Dabei kann eine Erweiterung abstrakt und damit ohne konkrete Ausprägungen definiert werden, um so eine bestimmte Erweiterbarkeit vorzubereiten. Listing 5.25 zeigt eine abstrakte Erweiterung B, die erst durch eine Erweiterung C eine konkrete Ausprägung erhält.

5.7 Fallbeispiel State Machine Language

In diesem Abschnitt zeige ich, wie eine Simulationssprache mit dem Ansatz DMX effizient entwickelt und ausgeführt werden kann. Ich beschränke mich auf eine spezifische Sprache als Fallbeispiel, die ausreichend komplex ist, um an ihr die Beschreibbarkeit ähnlich komplexer Sprachen zu zeigen. Eine komplexe DSL erlaubt die Beschreibung einer Menge von Problemlösungen in ihrer jeweiligen Domäne. Dafür stellt die Sprache Konstrukte bereit, die miteinander in Beziehung stehen. Erst durch die Verbindung der Konstrukte in einem konkreten Modell entsteht eine spezifische Problemlösung.

Als Beispielsprache wähle ich eine reduzierte UML-Zustandsmaschinensprache, die ich als Simple State Machine Language (SML) bezeichne. Die Sprache SML zeichnet sich durch eine Reihe von Besonderheiten aus, die sie als komplexe Beispielsprache charakterisieren. Ich gebe zunächst einen Überblick über die Sprache und erkläre ihre Besonderheiten. Die formale Definition der Sprache erfolgt im Anschluss durch eine Erweiterung von DBL.

5.7.1 Die Sprache SML

Eine Zustandsmaschine erlaubt die Beschreibung des Verhaltens eines aktiven Elementes in einem reaktiven System. Das Systemelement reagiert, je nach aktuellem Zustand, auf bestimmte Ereignisse, führt zugeordnete Aktionen aus und nimmt danach einen Folgezustand ein. Die Zustandsmaschine ist dabei einem aktiven Objekt zugeordnet. In SML erfolgt die Zuordnung durch die Definition einer Zustandsmaschine innerhalb einer aktiven Klasse. In UML kann eine Zustandsmaschine auch unabhängig von einem aktiven Objekt definiert werden. Dabei wird die Zustandsmaschine jedoch selbst zu einem aktiven Objekt, das auf Ereignisse reagiert. Die Zuordnung zu einer aktiven Klasse ist daher sinnvoll und stellt keine Einschränkung gegenüber Zustandsmaschinen aus UML dar.

Eine Zustandsmaschine besteht aus Zuständen und Transitionen. Einem Zustand können mehrere Transitionen zugeordnet sein, die jeweils einen Übergang von einem Zustand in einen Folgezustand festlegen. Ein Zustandsübergang erfolgt sobald ein Ereignis eintritt, das für die Transition angegeben ist. Eine Transition wird unterschieden in intern und extern. Bei einer internen Transition wird der aktuelle Zustand nicht verlassen und auch nicht neu betreten.

Als Folge eines Zustandsübergangs können Aktionen ausgeführt werden, die neue Ereignisse auslösen. Diese Aktionen werden ebenfalls als Teil einer Transition angegeben. Die möglichen Ereignisse auf die eine Transition reagieren kann sind:

1. Zeitereignisse,
2. Zustandsereignisse
3. und Signalempfangsereignisse.

Ein Zeitereignis enthält die Angabe einer Zeitspanne, die ausgehend von der aktuellen Modellzeit beim Betreten des Zustands vergehen muss. Bei einem Zustandsereignis muss die angegebene Bedingung erfüllt sein und bei einem Signalempfangsereignis muss ein Signal des angegebenen Typs empfangen werden.

Die Ausführung einer Zustandsmaschine beginnt mit einem ausgezeichneten Initialzustand. Sobald ein Ereignis für eine der ausgehenden Transitionen eintritt, werden die zugeordneten Anweisungen ausgeführt. Bei einer internen Transition ist die Ereignisbehandlung damit abgeschlossen.

Bei einer externen Transition wird der aktuelle Zustand verlassen und der Zielzustand wird betreten. Beim Betreten wird für alle ausgehenden Transitionen, die auf ein Zeitereignis reagieren, ein Zeitgeber gestartet, der ein Zeitereignis auslöst, sobald die angegebene Zeit vergangen ist. Beim Verlassen eines Zustands werden diese Zeitgeber gestoppt.

Sind Ereignisse an mehreren ausgehenden Transitionen zeitgleich erfüllt, so wird die Transition gewählt, die als erste angegeben ist. Auf diese Weise wird eine deterministische Auswahl bei Gleichzeitigkeit sichergestellt.

Besonderheiten

SML besitzt eine Reihe von Besonderheiten, die die Sprache als eine komplexe Sprache auszeichnen, die als Vorlage für ähnliche Sprachen dienen kann. SML enthält Sprach-

elemente, die Beziehungen untereinander definieren. Diese legen fest in welcher Form die Elemente miteinander kombiniert werden können.

Eine Beziehung zwischen einem Sprachelement A und einem Sprachelement B kann im Allgemeinen nur auf zwei Arten erfolgen: B ist in A enthalten oder A enthält eine Referenz auf B. In SML sind Zustände in einer Zustandsmaschine enthalten. Außerdem besitzen Transitionen eine Referenz auf einen Folgezustand. Zustände müssen somit identifizierbar sein.

Transitionen enthalten neben einem Verweis auf einen Folgezustand allgemeinsprachliche Elemente. Aktionen als Folge eines Zustandsübergangs werden mit Anweisungen ausgedrückt. Die Angabe eines Zeitereignisses oder Zustandsereignisses erfolgt mit einem Ausdruck. Diese allgemeinen Sprachelemente sind durch die Basissprache bereits verfügbar.

Der Laufzeitzustand einer Zustandsmaschine muss mehrfach instanziierbar sein, da eine Zustandsmaschine als Teil einer aktiven Klassen beschrieben wird und für jedes Objekt dieser Klasse eine Zustandsmaschine mit einem eigenen Laufzeitzustand existiert.

SML besitzt eine Ausführungssemantik, die durch Next-Event-Simulation beschrieben werden kann. Damit kann die Simulationsbasissprache für die Beschreibung der Semantik verwendet werden.

Zusammengefasst besitzt SML die folgenden Besonderheiten:

1. es gibt Sprachelemente, die in Beziehung zueinander stehen,
2. es gibt Sprachelemente, die über einen Bezeichner eindeutig identifizierbar sein müssen,
3. es gibt Sprachelemente, die allgemeinsprachliche Elemente enthalten,
4. es gibt Sprachelemente, die einen komplexen Laufzeitzustand besitzen, der für jede Instanz des Sprachelementes einen individuellen Zustand besitzt,
5. und es gibt Sprachelemente, die als Teil einer Simulation ausführbar sind.

5.7.2 Beispielanwendung Zugfiltersystem

Als Beispiel dient ein Modell eines einfachen Filtersystems, das Datenaktualisierungen zu einem Zug auf das zuletzt empfangene Datum in einem bestimmten Zeitintervall reduziert. Das Zeitintervall für den Filter startet sobald der erste Zug empfangen wird. Der Filter prüft nach Ende des Zeitintervalls, ob er noch eingeschaltet ist. Sollte der Filter während des Zeitintervalls ausgeschaltet werden, so wartet er bis er wieder eingeschaltet wird. Das Zeitintervall beginnt danach von vorn. Das Modell dient der Untersuchung des Systemverhaltens durch Beobachtung der Programmausgaben, die während der Simulation auftreten.

Listing 5.26 zeigt das Modell des Filtersystems in der um SML erweiterten Basissprache. Der Zugfilter wird als Objekt der aktiven Klasse Filter modelliert. Ein Zug ist ein Objekt des Signals Train und besitzt eine Nummer. Der Filter speichert den zuletzt empfangenen Zug im Attribut latestTrain. Zusätzlich definiert der Filter eine Kontrollvariable enabled, die festlegt, ob der Filter ein- (true) oder ausgeschaltet (false) ist.

Das Verhalten wird mit einer Zustandsmaschine festgelegt, die den initialen Zustand checking sowie die weiteren Zustände filtering und disabled enthält. Im Zustand checking gibt es zwei ausgehende Transitionen. Die erste Transition wartet auf ein Zustandsereignis. Falls der Filter ausgeschaltet wird, so wird in den Zustand disabled gewechselt. Die zweite Transition wartet auf ein Signal des Typs Zug und wechselt in den Zustand filtering. Der Zug wird dabei durch Aufruf der Methode update als latestTrain gespeichert.

Im Zustand filtering gibt es drei Transitionen. Die erste Transition wartet auf den Eintritt des Zeitereignisses 6, relativ zur aktuellen Modellzeit. Im Zustand filtering wird außerdem auf ein Zug-Signal gewartet und der Zustand wird beim Eintritt des Zustandsereignisses für einen ausgeschalteten Filter in den Zustand disabled verlassen. Der Zustand disabled kann nur durch Einschalten des Filters wieder verlassen werden.

Das Modell definiert mit der aktiven Klassen TrainArrival einen Ankunftsprozess für Züge. Zur Vereinfachung werden nur Züge mit der Nummer 2 erzeugt. Alle zwei Zeiteinheiten wird ein neues Zug-Signal erstellt und an das Filter-Objekt gesendet.

Der Anfangszustand des Modells und ein einfaches Experiment werden in der Methode main festgelegt. Es werden ein Filter-Objekt und ein Ankunftsprozess erzeugt und aktiviert. Danach wird der Filter nach 10 Zeiteinheiten eingeschaltet und nach 15 Zeiteinheiten wieder ausgeschaltet. Nach weiteren 10 Zeiteinheiten endet die Simulation. Bei der Ausführung des Modells werden Ausgaben erzeugt, die in Listing 5.27 gezeigt werden.

```
1  #import "ssm-language"
2  #import "../stdlib"
3
4  module singleTrainOnlyFilter;
5
6  signal Train(int number);
7
8  active class Filter {
9    Train latestTrain;
10   control boolean enabled = false;
11   ClassContent
12
13   state machine {
14     initial state checking {
15       when !enabled -> disabled
16       signal Train t do update(signal as Train); -> filtering
17     }
18     state filtering {
19       after 6 do publish(); -> checking
20       internal signal Train t do update(signal as Train);
21       when !enabled -> disabled
22     }
23     state disabled {
24       when enabled -> filtering
25     }
26   }
27
28   void publish() {
29     SystemOut.println("published train at " + time);
30     latestTrain = null;
31   }
```

```
32
33    void update(Train train) {
34      SystemOut.println("received update at " + time);
35      latestTrain = train;
36    }
37  }
38
39  active class TrainArrival {
40    Filter filter;
41
42    state machine {
43      initial state waiting {
44        after 2 do {
45          Train train = new Train();
46          train.number = 2;
47          send train to filter;
48        } -> waiting
49      }
50    }
51  }
52
53  void main() {
54    Filter filter = new Filter();
55    activate filter;
56
57    TrainArrival arrival = new TrainArrival();
58    activate arrival;
59    arrival.filter = filter;
60
61    advance 10;
62    filter.enabled = true;
63    advance 15;
64    filter.enabled = false;
65    advance  10;
66  }
```

Listing 5.26: Modell eines Zugfiltersystems in der um SML erweiterten Basissprache.

```
1  received update at 10.0
2  received update at 12.0
3  received update at 14.0
4  published train at 16.0
5  received update at 16.0
6  received update at 18.0
7  received update at 20.0
8  published train at 22.0
9  received update at 22.0
10 received update at 24.0
11 DefaultSimulation Experiment stopped at simulation time 35.0000.
```

Listing 5.27: Ausgabe bei Ausführung des Zugfiltersystems.

5.7.3 Filtersystem in DBL ohne die Erweiterung SML

Steht nur eine objektorientierte Sprache, wie z.B. die Basissprache DBL bereit, können Zustände und Transitionen als Objekte repräsentiert werden. Listing 5.28 zeigt

ein entsprechendes Modell in DBL. Der aktuelle Zustand wird als Wert der Variable currentState vom Typ State modelliert. Eine Transition ist ein Objekt der Klasse Transition, die als interne Transition markiert werden kann und einen Folgezustand nextState besitzt. Die Kontrollvariable signal repräsentiert einen stark vereinfachten Signalempfangspuffer, der für das Warten auf ein Zug-Signal verwendet wird.

```
#import "../stdlib"

module singleTrainOnlyFilterWithStatePattern;

class Train {
  int number;
}

active class Filter {
  Train latestTrain;
  control boolean enabled = false;

  // stark vereinfachter Signalpuffer (speichert nur das letzte Signal)
  control Object signal;

  actions {
    State initialState = new Checking(self);
    Transition initialTransition = new Transition(false, initialState);

    State currentState = initialTransition.nextState;
    Transition lastFired = initialTransition;

    while (currentState != null) {
      if (lastFired != null and !lastFired.internal) {
        currentState.enter();
      }

      lastFired = currentState.waitForEvent();
      if (lastFired != null) {
        currentState = lastFired.nextState;
      } else {
        currentState = null;
      }
    }
  }

  void send(Object newSignal) {
    signal = newSignal;
  }
  //...
}

class Transition {
  boolean internal;
  State nextState;

  new(boolean internal, State nextState) {
    self.internal = internal;
    self.nextState = nextState;
  }
```

```
51 }
52
53 interface State {
54   void enter();
55   Transition waitForEvent();
56 }
```

Listing 5.28: Modell eines Zugfiltersystems in der Basissprache unter Verwendung von Objekten für Zustände.

Die Ereignisschleife wird ausgeführt, solange der aktuelle Zustand einen gültigen Wert hat. Das Interface State definiert eine Reihe von Methoden für einen Zustand, die für die Modellierung der Ereignisschleife verwendet werden. Falls keine interne Transition ausgelöst wurde, so wird der neue Zustand zunächst durch Aufruf der Methode enter betreten. Danach wird auf den Eintritt eines Ereignisses im aktuellen Zustand gewartet, in dem die Methode waitForEvent aufgerufen wird.

Für jeden Zustand gibt es eine entsprechende Klasse, die das Interface State implementiert. Als Beispiel wird die Klasse für den Zustand filtering in Listing 5.29 gezeigt. Die Klasse Filtering besitzt die Attribute context für den Zugriff auf das Filter-Objekt und timer für das Warten auf das Zeitereignis. Nach Betreten des Zustands wird der Timer gestartet. Der Timer läuft nach der definierten Zeit ab und setzt die Kontrollvariable expired auf den Wert true.

```
1  class Filtering implements State {
2    Filter context;
3    Timer timer;
4
5    new(Filter context) {
6      self.context = context;
7    }
8
9    void enter() {
10     timer = new Timer(6);
11     activate timer;
12   }
13
14   Transition waitForEvent() {
15     wait until timer.expired
16       or context.signal != null and context.signal instanceof Train
17       or !context.enabled;
18
19     // Aktionen
20     if (timer.expired) {
21       timer.expired = false;
22       context.publish();
23       return new Transition(false, new Checking(context));
24     }
25     if (context.signal != null and context.signal instanceof Train) {
26       context.update(context.signal as Train);
27       context.signal = null;
28       return new Transition(true, self);
29     }
30     if (!context.enabled) {
31       return new Transition(false, new Disabled(context));
32     }
```

```
33      return null;
34    }
35  }
36
37  active class Timer {
38    int duration;
39    control boolean expired = false;
40
41    new(int duration) {
42      self.duration = duration;
43    }
44
45    actions {
46      advance duration;
47      expired = true;
48    }
49  }
```

Listing 5.29: Implementierung der State-Klasse für den Zustand filtering.

Das Warten auf eines der möglichen Ereignisse wird mit der Anweisung wait until beschrieben. Sobald ein Ereignis vorliegt, muss geprüft werden, um welches Ereignis es sich handelt. Danach wird die entsprechende Aktion ausgeführt und es wird eine Transition für den Folgezustand zurückgegeben. Das Transition-Objekt trägt die Information, ob eine interne oder eine externe Transition ausgelöst wurde.

5.7.4 Erweiterungsdefinition SML

Die Spracherweiterung SML besteht aus vier Erweiterungsdefinitionen: Signaldefinition (SignalDefinition), Signalzugriff (SignalAccess), Sendeanweisung (SendStatement) und Zustandsmaschine (StateMachine).

Signaldefinition

Eine Signaldefinition erweitert das Basiskonzept Klasse und definiert Regeln für die Angabe eines Bezeichners und einer Liste von Attributen (siehe Listing 5.30). Die Semantikdefinition erfolgt als Abbildung auf eine Klasse mit einem Konstruktor. Dabei wird die ForEach-Erweiterung eingesetzt, um über die Attribute zu iterieren. Die importierten Dateien stdlib und stdx definieren eine Standardbibliothek mit Binding-Definitionen und einige Standarderweiterungen (vollständige Definition siehe Anhang in den Listings A.6 und A.7).

```
1  #import "../dbl"
2  #import "../stdlib"
3  #import "../stdx"
4
5  module ssm;
6
7    extension SignalDefinition extends dbl Class {
8      start SignalDefinition;
9      SignalDefinition -> "signal" name:ID EnclosedAttributes ";";
10     EnclosedAttributes -> "(" Attributes ")";
11     Attributes -> attributes : list Variable MoreAttributes;
```

```
12      MoreAttributes -> "," Attributes;
13      MoreAttributes -> ;
14    }
15    semantics for SignalDefinition {
16      expand "class " name " {";
17      foreach (Variable attribute in attributes) {
18        expand attribute ";";
19      }
20
21      expand "new(" attributes ") {";
22      foreach (Variable attribute in attributes) {
23        expand "self." attribute.getName()
24          "=" attribute.getName() ";";
25      }
26      expand "}";
27
28      expand "}";
29    }
```

Listing 5.30: Erweiterungsdefinition für Signaldefinition.

Zustandsmaschine

Eine Zustandsmaschine soll im Kontext einer Klasse definiert werden können. Die Erweiterungsdefinition erfolgt deshalb durch Erweiterung des Basiskonzeptes ClassContentExtension. Dieses Basiskonzept besitzt keine konkrete Ausprägung und dient lediglich der Angabe von Erweiterungen im Kontext einer Klasse, die an dieser Stelle für die Einführung von Erweiterungen vorbereitet wurde.

Die Semantikdefinition erfolgt durch eine Abbildung auf einen actions-Teil, der eine Schleife für die Ereignisbehandlung enthält. Dabei kann die in Abschnitt 5.7.3 vorgestellte Implementierung einer Zustandsmaschine in der Basissprache für die Abbildung nicht komplett übernommen werden. Das Problem ist, dass der Kontext von Ausdrücken und Anweisungen, die in der Zustandsmaschine angegeben sind, angepasst werden müsste. Diese Anpassung stellt ein schwieriges Problem dar. Als Teil der Abbildung müsste der abstrakte Syntaxbaum jeder Anweisung analysiert und die Abbildung von Ausdrücken um eine Variable ergänzt werden, die den Kontext anpasst.

Um das Problem der Kontextanpassung zu vermeiden, werden Ausdrücke und Anweisung auf Methoden im Kontext der Klasse, zu der die Zustandsmaschine gehört, abgebildet. Eine Zustand wird dabei durch eine ganze Zahl repräsentiert. Die Ereignisbehandlung prüft den aktuellen Zustand und ruft danach die entsprechende Methode auf. Das Ziel der Abbildung ist vereinfacht in Listing 5.31 dargestellt. Für das Prüfen des aktuellen Zustands wird eine Switch-Anweisung verwendet, die je nach Wert die passende Methode für das Warten auf Ereignisse aufruft.

```
1 class Transition {
2   boolean internal;
3   int nextState;
4
5   new(boolean internal, int nextState) {
6     self.internal = internal;
7     self.nextState = nextState;
8   }
```

```
}

active class Filter {
  Train latestTrain;
  control boolean enabled = false;

  control Object signal;
  Timer timer1;

  // Zustandstabelle:
  // 0 -> checking
  // 1 -> filtering
  // 2 -> disabled
  // -1 -> final

  actions {
    int initialState = 0;
    Transition initialTransition = new Transition(false, initialState);

    int currentState = initialTransition.nextState;
    Transition lastFired = initialTransition;

    while (currentState != -1) {
      if (lastFired != null and !lastFired.internal) {
        enter(currentState);
      }

      switch (currentState) {
        case 0:
          lastFired = checking_wait();
          break;
        case 1:
          lastFired = filtering_wait();
          break;
        case 2:
          lastFired = disabled_wait();
          break;
      }

      if (lastFired != null) {
        currentState = lastFired.nextState;
      } else {
        currentState = -1;
      }
    }
  }
  // ...

  Transition filtering_wait() {
    wait until timer1.expired
      or signal != null and signal instanceof Train
      or !enabled;

    if (timer1.expired) {
      timer1.expired = false;
      publish();
      return new Transition(false, 0);
```

```
66      }
67      if (signal != null and signal instanceof Train) {
68        update(signal as Train);
69        signal = null;
70        return new Transition(true, 1);
71      }
72      if (!enabled) {
73        return new Transition(false, 2);
74      }
75      return null;
76    }
77  }
```

Listing 5.31: Ziel der Abbildung einer Zustandsmaschine.

Die Datei, die die Erweiterung definiert, enthält neben Erweiterungsdefinitionen weitere Klassen, die eine Laufzeitbibliothek für die Zustandsmaschine bilden. Dazu gehören die Klassen Transition und Timer. Bei der Ersetzung der entsprechenden Erweiterungsinstanzen sind die notwendigen Klassen damit bereits vorhanden.

Die Syntaxdefinition ist in Listing 5.32 und die Semantikdefinition ist ausschnittsweise in Listing 5.33 dargestellt. Die Abbildung erfolgt zunächst auf eine Variable für das aktuelle Signal und einen actions-Teil mit einer Ereignisbehandlung. Dabei wird mehrfach über die Zustände iteriert, um die Aufrufe an die Methoden für das Betreten des aktuellen Zustands und für das Warten auf ein Ereignis abzubilden. Nach dem der die Zustandsmaschine durch den actions-Teil ersetzt wurde, werden danach die entsprechenden Methoden hinzugefügt. Die Implementierung der Methoden ist hier als Teil der Abbildung nicht angegebenen und wird nur vereinfacht dargestellt.

Darüber hinaus wird zur Vereinfachung eine expand-Anweisung verwendet, bei der der Kontext beliebig durch Angabe eines Metaobjektes verändert werden kann. Die konkrete Syntax der Basiskonzepte wird dabei nach der konkreten Syntax des angegebenen Metaobjektes hinzugefügt.

```
1  #import "../dbl"
2  #import "../stdlib"
3  #import "../stdx"
4
5  module ssmsyntax;
6
7  extension StateMachine extends dbl ClassContentExtension {
8    start StateMachine;
9
10   StateMachine -> "state" "machine" "{" ManyStates "}";
11   ManyStates -> ;
12   ManyStates -> states : list State ManyStates;
13
14   State -> InitialModifier "state" name:ID "{"
15     ManyTransitions
16   "}";
17   InitialModifier -> initial:"initial";
18   InitialModifier -> ;
19
20   ManyTransitions -> ;
21   ManyTransitions -> outgoing : list Transition ManyTransitions;
22
```

```
23   Transition -> InternalTransition;
24   Transition -> ExternalTransition;
25
26   InternalTransition -> internal:"internal" Trigger;
27   ExternalTransition -> Trigger "->" TargetState;
28
29   Trigger -> "when" condition:Expression Effects;
30   Trigger -> "signal" messageVariable:Variable Effects;
31   Trigger -> "after" timeDuration:Expression Effects;
32   Effects -> ;
33   Effects -> "do" effects : SimpleStatement;
34
35   TargetState -> target:$$State;
36   TargetState -> stop:"stop";
37 }
```

Listing 5.32: Syntaxdefinition Zustandsmaschine.

```
1  #import "../dbl"
2  #import "../stdlib"
3  #import "../stdx"
4  #import "ssm-language-statemachine-syntax"
5
6  module ssmsyntax;
7
8  class Transition {
9    boolean _internal;
10   int nextState;
11
12   new(boolean _internal, int nextState) {
13     self._internal = _internal;
14     self.nextState = nextState;
15   }
16 }
17
18 semantics for StateMachine {
19   expand
20   "control Object currentSignal;"
21   "actions {"
22     "int initialState = 0;"
23     "Transition initialTransition = new Transition(false, initialState);"
24
25     "int currentState = initialTransition.nextState;"
26     "Transition lastFired = initialTransition;"
27
28     "while (currentState != -1) {"
29       "if (lastFired != null and !lastFired._internal) {"
30         "switch (currentState) {";
31
32   int i=0;
33   foreach (State state in states) {
34     expand "case " i ":"
35       "enter_" i "();"
36       "break;";
37     i=i+1;
38   }
39
```

```
40    expand
41          "}"
42        "}"
43        "switch (currentState) {";
44
45    i=0;
46    foreach (State state in states) {
47      expand "case " i ":"
48        "lastFired = wait_" i "();"
49        "break;";
50      i=i+1;
51    }
52
53    expand
54        "}"
55        "if (lastFired != null) {"
56          "currentState = lastFired.nextState;"
57        "} else {"
58          "currentState = -1;"
59        "}"
60      "}"
61    "}";
62
63    i=0;
64    foreach (State state in states) {
65      expand "void enter_" i "() {"
66        "}"
67        after self;
68
69      expand "Transition wait_" i "() {"
70        "}"
71        after self;
72
73      i=i+1;
74    }
75  }
```

Listing 5.33: Semantikdefinition Zustandsmaschine.

Signalempfang – Zugriff mit Schlüsselwort

Die Grammatikregel Trigger führt zwar eine Variable für die Definition eines Signalempfangs ein, jedoch kann in den Anweisungen, die die Regel Effects festlegt, nicht auf die Variable zugegriffen werden. Dazu müsste, ausgehend von den Anweisungen, eine Namensauflösung für die Signalvariable definiert werden. Die Definition von Namensauflösungen liegt jedoch nicht im Fokus dieser Arbeit.

Das Problem wird durch die Einführung eines Schlüsselwortes für die Signalvariable vermieden (siehe Listing 5.34). Die Erweiterungsdefinition SignalAccess ist eine Erweiterung für einen Ausdruck, der auf eine Variable currentSignal abgebildet wird, die bei der Abbildung einer Transition mit einem Signalempfang stets zugreifbar ist.

```
1  #import "../dbl"
2  #import "../stdlib"
3  #import "../stdx"
```

```
4
5  module ssm;
6
7  extension SignalAccess extends dbl L1Expr {
8    start SignalAccess;
9    SignalAccess -> "signal";
10 }
11 semantics for SignalAccess {
12   expand "currentSignal";
13 }
```

Listing 5.34: Erweiterungsdefinition SignalAccess.

Signalempfang – Zugriff über Variable

Eine weitere Möglichkeit, die jedoch nur syntaktisch unterstützt wird, ist die Einführung einer separaten Erweiterungsdefinition für eine Signal-Transition durch die Erweiterung SignalTrigger, die in der Erweiterungsdefinition für die Zustandsmaschine verwendet wird (siehe Listing 5.35). Durch Erweiterung des Basiskonzeptes für einen lokalen Gültigkeitsbereich, werden die Signalvariable und die Anweisungen auf Basis des Konzeptes LocalScopeStatement definiert, für das die Namensauflösung durch die Basissprache bereits festgelegt ist. Die Erweiterungsdefinition SignalTrigger verwendet dabei für die Signalvariable das Attribut statements des Basiskonzeptes LocalScopeStatement. Die Anweisungen für die Ereignisbehandlung verwenden ebenfalls dieses Attribut.

Für die Erweiterung SignalTrigger darf jedoch keine Semantik als Abbildung auf Basiskonzepte definiert werden, da die Abbildung als Teil der Semantikdefinition der Zustandsmaschine festgelegt werden muss. In diesem Fall wäre die Einführung einer reinen syntaktischen Erweiterung notwendig.

```
1  extension SignalTrigger extends dbl LocalScopeStatement {
2    start SignalTrigger;
3
4    SignalTrigger -> "signal" statements:list Variable Effects;
5    Effects -> ;
6    Effects -> "do" statements:list SimpleStatement;
7  }
8
9  extension StateMachine extends dbl ClassContentExtension {
10   start StateMachine;
11
12   StateMachine -> "state" "machine" "{" ManyStates "}";
13   //...
14   Trigger -> signalTrigger:SignalTrigger;
15 }
```

Listing 5.35: Syntaxdefinition für eine separate SignalTransition.

Sendeanweisung

Die Anweisung für das Senden eines Signals ist in Listing 5.36 dargestellt. Die Abbildung erfolgt auf den Aufruf der Methode send, die für die Klasse, die eine Zustandsma-

schine enthält, durch die Erweiterungsdefinition einer Zustandsmaschine zur Klasse hinzugefügt wird.

```
#import "../dbl"
#import "../stdlib"
#import "../stdx"

module ssmsignalstatement;

extension SendStatement extends dbl SimpleStatement {
  start SendStatement;
  SendStatement -> "send" signal:Expression
    "to" target:Expression ";";
}
semantics for SendStatement {
  expand target ".send(" signal ");";
}
```

Listing 5.36: Erweiterungsdefinition für eine Sendeanweisung.

5.8 Vergleich mit ähnlichen Ansätzen

5.8.1 Metamodellbasierte Ansätze

Bei einem metamodellbasierten Ansatz stellt das Metamodell das zentrale Artefakt einer Sprachdefinition dar. Andere Sprachaspekte werden unter Bezugnahme auf Elemente des Metamodells definiert. Für die Definition einer DSL mit einem Metamodell gibt es verschiedene Ansätze, die sich in den verfügbaren Metasprachen und in den bereitgestellten Sprachwerkzeugen unterscheiden. Ich stelle die Ansätze zunächst kurz vor und vergleiche diese danach zusammen in Bezug auf charakteristische Merkmale, die in jedem metamodellbasierten Ansatz vorhanden sind.

Scheidgen [62] beschreibt einen Ansatz, der die Metasprache CMOF für ein Metamodell, TSL für eine Grammatik und UML-Aktivitätsdiagramme für eine Ausführungssemantik verwendet. Scheidgen zeigt die Anwendbarkeit seines Ansatzes am Beispiel der Sprache System Description Language (SDL). SDL ist eine komplexe Sprache, mit der die Struktur und das Verhalten eines Systems beschrieben werden können.

Soden [70] beschreibt in seiner Arbeit einen ähnlichen Ansatz. Er setzt M3Actions als gemeinsame Metasprache für die Definition eines Metamodells und einer Ausführungssemantik ein. Die Sprache M3Actions enthält Konzepte der Metasprache Ecore für die Definition der abstrakten Syntax und UML-Aktivitätsdiagramme für die Definition der Ausführungssemantik. Für jede Operation in einer Metaklasse kann eine UML-Aktivität erstellt werden, die das Verhalten dieser Operation festlegt.

Sadilek [57] verwendet als Metasprache für das Metamodell ECore und setzt TSL für die Grammatik ein. Für die Definition der Ausführungssemantik erlaubt sein Ansatz den Einsatz verschiedener bestehender Programmiersprachen. Er entwickelt das Framework EProvide mit dem auf eine Metamodellinstanz in einer beliebigen anderen Sprache zugegriffen und diese verändert werden kann. Auf diese Weise kann die Ausführungssemantik einer DSL operational definiert werden.

In den Arbeiten von Scheidgen, Soden und Sadilek wird die Ausführungssemantik mit einer operationalen Semantik beschrieben. Sie entwickeln in ihren Arbeiten Sprachwerkzeuge, die sich für die praktische Entwicklung einer DSL sofort einsetzen lassen. In ihren Anwendungsfällen verwenden sie jedoch keine Simulationssprache. Sie erklären lediglich die prinzipielle Möglichkeit zur Entwicklung einer solchen Sprache.

Im Gegensatz dazu konzipieren Prinz, Møller-Pedersen und Fischer [56] einen ähnlichen Ansatz und beschreiben seine Anwendung am Beispiel einer Simulationssprache. Die Arbeit bleibt jedoch auf einer konzeptionellen Ebene. Sprachwerkzeuge, die den Ansatz implementieren, stehen nicht bereit. Der Ansatz erlaubt den Einsatz einer beliebigen Strukturbeschreibungssprache für das Metamodell und setzt eine Strukturelle Operationale Semantik für die Definition einer Ausführungssemantik ein.

Das Framework Xtext implementiert einen Ansatz, der vergleichbar mit dem Ansatz von Scheidgen ist. Xtext erlaubt die Definition der konkreten Syntax einer DSL mit einer attributierten Grammatik. Darüber hinaus ist es möglich ein Metamodell aus einer Xtext-Grammatik zu erzeugen, um so den Aufwand für die Definition des Metamodells zu vermeiden. Xtext enthält keine Metasprache für die Definition einer Ausführungssemantik.

Das Meta Programming System (MPS) ist ähnlich zu den bisher vorgestellten Ansätzen, unterscheidet sich jedoch in der Definition der konkreten Syntax und der Ausführungssemantik grundlegend. MPS ist ein Framework mit dem sich alle Aspekte einer DSL beschreiben lassen. Auf der Basis von Teilaspektbeschreibungen wird eine DSL-spezifische Entwicklungsumgebung abgeleitet, die vergleichbar mit integrierten Entwicklungsumgebungen für Programmiersprachen ist. In MPS wird die konkrete Syntax nicht mit einer Grammatik, sondern als Projektion einer Metamodellinstanz auf einen Text beschrieben. Damit ist das Parsen der Eingabe nicht mehr notwendig und es können keine Mehrdeutigkeiten in der Definition der konkreten Syntax auftreten. Die Modellierung mit einem projektionalen Editor ist jedoch ungewohnt. Der Modellierer arbeitet auf dem abstrakten Syntaxbaum und nimmt an diesem Veränderungen vor. Das Modell kann nicht frei durch die Eingabe von Text an beliebiger Stelle im Editor entwickelt werden und wird auch nicht als Text serialisiert. Damit lassen sich keine textbasierten Werkzeuge mehr einsetzen. Ein projektionaler Editor bietet jedoch den Vorteil der Kombination verschiedener Spracherweiterungen, ohne das hierbei syntaktische Konflikte auftreten können.

Die vorgestellten metamodellbasierten Ansätze besitzen viele Ähnlichkeiten. Ich vergleiche die Ansätze deshalb zusammen mit DMX und nehme Bezug auf ihre charakteristischen Merkmale.

Getrennte Definition von Teilaspektbeschreibungen

Bei den vorhandenen Ansätzen können allgemeinsprachliche Konzepte für jeden Teilaspekt nur separat beschrieben und damit auch nur separat wiederverwendet werden. Der DSL-Entwickler ist selbst dafür verantwortlich, dass die Teilaspektdefinitionen eines Sprachkonzeptes die vorhandenen allgemeinsprachlichen Konzepte in allen Aspektbeschreibungen konsistent spezialisieren und kein Widerspruch entsteht. Dabei kann nur das Metamodell durch das in CMOF und Ecore vorhandene Vererbungskonzept spezialisiert werden. Für eine Grammatik ist das jedoch nicht möglich.

Die Ausführungssemantik kann nur durch eine Aufteilung auf Operationen von Metaklassen wiederverwendbar definiert werden.

Im Gegensatz dazu erfolgt in DMX die Definition der Teilaspekte eines Sprachkonzeptes zusammen in Form einer Erweiterung. Auf diese Weise lassen sich allgemeinsprachliche Konzepte spezialisieren und auch leicht in eine Erweiterung integrieren und damit wiederverwenden.

Wiederholung ähnlicher Strukturen in Metamodell und Grammatik

Ähnliche Strukturen müssen im Metamodell und in der Grammatik separat definiert und miteinander in Beziehung gesetzt werden. Dies führt in vielen Fällen zu einem zusätzlichen Aufwand, da sich ähnliche Strukturen wiederholen. Die Entwicklung der ersten Entwürfe einer DSL wird damit schwieriger, da es in dieser Phase potenziell viele Änderungen an der Sprache gibt. In DMX wird das Metamodell aus der Grammatik abgeleitet. Damit erhöht sich die Effizienz in der Entwicklung einer DSL.

Verzögerte Bereitstellung von Sprachwerkzeugen

Sprachwerkzeuge müssen bei den vorhandenen Ansätzen manuell aus einer Sprachbeschreibung erzeugt werden. Das Werkzeug muss danach in ein ausführbares Programm übersetzt werden und steht erst dann für die Entwicklung von Modellen in einer DSL zur Verfügung. Damit ist die Entwicklung einer DSL ineffizienter, da Sprachwerkzeuge nicht sofort vorhanden sind. Dagegen erfolgt die Bereitstellung des Modelleditors bei DMX unmittelbar während eine DSL als Spracherweiterung definiert wird. Der Modelleditor analysiert die Sprachbeschreibung zur Laufzeit und stellt die gewohnte Eingabeunterstützung passgenau für das neue Konzept bereit. Ebenso steht der Transcompiler für eine DSL unmittelbar bereit, da die Semantik als Abbildung auf die Basissprache definiert wird und für die Basissprache bereits ein Compiler vorhanden ist.

Ausführungssemantik von Simulationssprachen ist nicht effizient beschreibbar

Die Semantik wird in den Ansätzen von Scheidgen, Soden, Sadilek und Prinz und Fischer als operationale Semantik definiert. Dabei handelt es sich um eine Ausführungssemantik, die eine schrittweise Veränderung des Programmzustands festlegt. Der Programmzustand ist ebenfalls durch das Metamodell beschrieben.

Eine operationale Semantik ist geeignet eine Sprache unabhängig von der Ausführung auf einer konkreten Maschine zu beschreiben und erlaubt beweisbare Aussagen zu bestimmten Eigenschaften der Sprache. Zustände werden abstrakt und ohne Bezug zu den tatsächlichen Stukturen einer konkreten Maschine definiert. Die Laufzeit hängt aber von einer Abbildung auf diese konkreten Strukturen ab. Die Anforderungen von Simulationssprachen in Bezug auf Laufzeiteffizienz können mit einer solchen Semantik nicht erreicht werden.

Ein Beispiel sind pseudo-parallele Prozesse, die jeweils eine Reihe von Anweisungen nacheinander abarbeiten. Der Laufzeitzustand besteht hier aus einer Menge von Prozessen und je Prozess aus einem Verweis auf die nächste auszuführende Anweisung. In einer operationalen Semantik ist eine Anweisung als ein Objekt definiert. Je nach Art

der Anweisung hat das Objekt eine bestimmte Semantik. Die operationale Semantik bestimmt das nächste Anweisungsobjekt und interpretiert es.

Auf einer konkreten Maschine können die Anweisungen einer solchen Sprache dagegen direkt auf eine Sequenz von Maschinenbefehlen abgebildet werden. Die nächste auszuführende Anweisung entspricht einem Verweis auf einen Maschinenbefehl und befindet sich in einem Register, einem speziellen Speicher mit einer sehr kurzen Zugriffszeit. Änderungen am Laufzeitzustand sind damit sehr effizient möglich.

Laufzeiteffiziente Simulationssprachen besitzen deshalb eine transformationale Semantik, implementiert durch einen Compiler, oder sie sind Teil von Programmiersprachen, für die ebenfalls ein Compiler existiert. Die Beschreibung einer transformationalen Semantik ist mit DMX möglich.

5.8.2 Erweiterungsbasierte Ansätze

Bei der erweiterungsbasierten Definition einer DSL können zwei Arten von Ansätzen unterschieden werden. Auf der einen Seite gibt es Ansätze, die einen Präprozessor verwenden. Eine Erweiterung wird hier in einer Präprozessorsprache oder in einer Basissprache definiert. Für die Ausführung eines Programms werden Erweiterungsinstanzen durch den Präprozessor in die Basissprache abgebildet. Diese Ansätze besitzen kaum Unterstützung durch DSL-spezifische Sprachwerkzeuge. Zingaro [86] geht in seiner Arbeit detailliert auf diese Ansätze ein und vergleicht sie miteinander.

Auf der anderen Seiten gibt es metamodellbasierte Ansätze zur Spracherweiterung. Die Konzepte einer Basissprache werden hier für die Definition einer DSL eingebunden werden. DSL-spezifische Sprachwerkzeuge können aus einer Sprachbeschreibung automatisch abgeleitet werden.

Präprozessor-basiert

Präprozessor-basierte Ansätze erlauben nur die Definition einfacher isolierter Sprachkonstrukte als Erweiterung. Komplexe Konstrukte, die miteinander in Beziehung stehen, lassen sich mit diesen Ansätzen nicht beschreiben. Die Unterstützung durch Sprachwerkzeuge ist kaum vorhanden.

Der Java Syntactic Extender (JSE) [4] ist ein Präprozessor für die Programmiersprache Java. Erweiterungen sind hier auf wenige Formen beschränkt, deren Syntax vordefiniert ist: Funktionsaufrufe, Macros und Anweisungsmacros. Die Erweiterungen müssen immer mit einem Bezeichner beginnen und mit einer bestimmten festen Zeichenfolge enden. Erweiterungen können aus Basiskonstrukten zusammengesetzt sein. Sie können jedoch nicht auf Basiskonstrukte verweisen. JSE erlaubt in einer Erweiterung die Angabe des Typs eines enthaltenen Basiskonstrukts. So ist es möglich eine For-Each-Anweisung zu definieren, die aus Ausdrücken und Anweisungen bestehen muss.

Camlp4 [20] ist ein Präprozessor für die Multiparadigmensprachen Ocaml. Im Gegensatz zu JSE erfolgt die Definition einer Erweiterung auf der abstrakten Syntax der Basissprache Ocaml. Die Grammatik von Ocaml kann durch neue Regeln erweitert werden und es können bestehende Regeln verändert oder entfernt werden. Durch die Möglichkeit die abstrakte Syntax der Basissprache zu ergänzen, sind auch Erweiterungen von Ausdrücken unter Angabe des Vorrangs und der Assoziativität von Opera-

toren möglich. Eine Einschränkung besteht durch die LL-Grammatik der Basissprache Ocaml. Erweiterungen sind ebenfalls auf eine LL-Grammatik beschränkt.

Die Erweiterbarkeit der Simulationssprache SLX [34] beschränkt sich auf die Angabe von regulären Ausdrücken. Damit lassen sich lediglich reguläre Sprachen als Erweiterung der Basissprache von SLX definieren. Mit SLX ist es nicht möglich Verweise zwischen Erweiterungen festzulegen, um so komplexe Sprachkonzepte zu beschreiben. Der Modelleditor bietet jedoch eine einfache Unterstützung für Erweiterungen, in dem die Syntaxteile von Erweiterungen in einem SLX-Programm nach erfolgreicher Übersetzung hervorgehoben werden.

Metamodell-basiert

Das Framework Xtext [36] stellt die Definition der Basissprache Xbase bereit, deren Konzepte sich in eine DSL integrieren lassen. Xtext erlaubt jedoch nicht die Erweiterung der Konzepte von Xbase. Eine DSL kann in Xtext keinen neuen Ausdruck definieren, der zusammen mit bestehenden Xbase-Ausdrücken eingesetzt werden kann. Die Konzepte von Xbase sind außerdem auf Ausdrücke und Anweisungen beschränkt. Es gibt keine Konzepte für Strukturdefinitionen. Mit Xtext kann ein textueller Editor generiert werden.

Das Meta Programming System (MPS) [54] besitzt im Gegensatz zu Xtext eine erweiterbare Basissprache, die als Base Language (BL) bezeichnet wird. Die Definition einer DSL beginnt mit der Festlegung einer abstrakten Syntax in Form eines Metamodells. Danach erfolgt die Definition der konkreten Syntax für die Konzepte des Metamodells.

Der Modelleditor, der von MPS für eine DSL bereitgestellt werden, arbeitet grundsätzlich anders als Editoren anderer Ansätze für textuelle DSLs. Die Eingabe erfolgt nicht als frei veränderbarer Text, wie bei einem Texteditor. Der Modelleditor arbeitet stattdessen auf dem abstrakten Syntaxbaum und erlaubt dem Modellierer die Instanzierung von Konzepten des Metamodells, ausgehend von einem Wurzelelement. Dabei wird jedes Objekt auf Basis der konkreten Syntax auf eine Text-Repräsentation projeziert. Verweise auf andere enthaltene oder referenzierte Objekte erscheinen zunächst als Platzhalter. Der Modellierer navigiert zwischen Textteilen, die mit Werten befüllt werden können wie in einem Baum. Dadurch ist das Parsen eines Textes und eine nachgelagerte Instanziierung des Metamodells nicht notwendig.

Der Modelleditor ist jedoch ungewohnt in der Bedienung. Ein Modellierer kann den Text nicht vollkommen frei verändern und muss seine Anpassungen immer entlang von Baumstrukturen des Metamodells vornehmen. Nachdem an einer Stelle eine bestimmtes Objekt platziert wurde, werden die statischen Teile der konkreten Syntax des Konzeptes unveränderbar eingefügt. Dazwischen befinden sich veränderbare Textstellen, die mit einfachen Werten, wie z.B. einem Bezeichner oder einer Zahl, oder mit komplexen Objektstrukturen befüllt werden können. So beginnt z.B. eine Klasse immer mit dem Text class, gefolgt von einem beliebigen Bezeichner, einer öffnenden geschweiften Klammer, einer beliebigen Anzahl von Objekten für Attribute und Methoden, sowie einer schließenden geschweiften Klammer.

Die Basissprache BL kann um neue Konzepte erweitert und in eine DSL eingebunden werden. Außerdem ist es möglich Referenzen zwischen Konstrukten einer DSL

und auch auf Konstrukte der Basissprache zu definieren. MPS erlaubt die Generierung eines textuellen Editors nachdem die Definition einer DSL abgeschlossen ist. MPS ist ein interaktives Framework, das die Definition verschiedener Aspekte einer DSL erlaubt.

Ein Modelleditor ist wohl in Xtext als auch in MPS nicht unmittelbar verfügbar. Bevor der Editor ausgeführt werden kann, muss die Software des Editors zunächst erzeugt und danach übersetzt werden. Die Generierung muss jedesmal bei Änderungen an der Sprachdefinition erneut ausgeführt werden. Damit wird die effiziente Entwicklung einer DSL erschwert.

Mit MPS lassen sich auch andere Basissprachen erweiterbar definieren. Eine erweiterbare Sprache, die C als Basissprache verwendet und mit MPS entwickelt wird, ist die Sprache embeddr [79], die eine Reihe Spracherweiterungen bereitstellt und um weitere Erweiterungen ergänzt werden kann. embeddr eignet sich durch die bereitgestellten Spracherweiterungen für eine Modellierung von eingebetteten Systemen, die sicherer ist als eine ausschließliche Verwendung der Sprache C. embeddr stellt dazu unter anderem eine Sprache für die zustandsorientierte Verhaltensmodellierung bereit. Hier wird von embeddr auch die Verifikation bestimmter Eigenschaften einer Zustandsmaschine unterstützt. Da embeddr auf MPS basiert, erfolgt die Modellierung ebenfalls mit einem projezierenden Editor.

5.8.3 Sprachintegrierte grammatikbasierte Ansätze

Als weitere verwandte Arbeiten sind Parser-Kombinatoren zu nennen. Dabei handelt es sich um sprachintegrierte grammatikbasierte Ansätze, die z.B. für Scala [47] und Java vorhanden sind. Die Definition einer Grammatik erfolgt hier in der Basissprache selbst. Dazu werden die Konzepte der Basissprache so eingesetzt, dass eine kontextfreie Grammatik angegeben werden kann. Eine Grammatik wird inklusive semantischer Aktionen in der Sprache selbst beschrieben. Sprachinstanzen können jedoch nur als einfache Zeichenketten in der Ausgangssprache angegeben werden und sind nicht in die Ausgangssprache integriert. Die Zeichenkette wird einem Parser übergeben, der sie auf Basis der angegebenen Grammatik analysiert und die semantischen Aktionen ausführt. Der Vorteil von Parser-Kombinatoren ist die Möglichkeit die Semantik in der Sprache des Kombinators anzugeben. Es lassen sich jedoch keine bestehenden Elemente der Basissprache in einer Erweiterung wiederverwenden. Darüber hinaus lassen sich nur die Sprachwerkzeuge der Basissprache verwenden, die für Erweiterungen keine besondere Unterstützung bieten.

6 Zusammenfassung

Zum Schluss blicke ich auf die Beiträge der Arbeit zurück und gebe einen Ausblick für die Zukunft.

In dieser Dissertation habe ich mich mit der effizienten Entwicklung und Ausführung von Simulationssprachen für domänenspezifische Anwendungsfelder beschäftigt. Für die effiziente Entwicklung habe ich eine Simulationsbasissprache definiert, die sich flexibel erweitern lässt und die notwendigen prozessorientierten Modellierungskonzepte für laufzeiteffizient ausführbare Simulationen als Basiskonzepte enthält. Für die effiziente Ausführung habe ich eine Methode für die Abbildung von Prozesskontextwechseln auf ein C++-Programm vorgestellt, die eine besonders laufzeiteffiziente Ausführung von Kontextwechseln in einer Simulation erlaubt. Die effiziente Entwicklung domänenspezifischer Erweiterungen der Basissprache wird durch den Ansatz DMX ermöglicht. Ich habe den Ansatz unabhängig von der Domäne Simulation beschrieben, da die Konzepte auch auf andere Basissprachen übertragbar sind. Dazu habe ich Voraussetzungen für die Übertragbarkeit des Ansatzes auf andere Basissprachen identifiziert. Die Anwendbarkeit für die Entwicklung einer komplexen Sprache habe ich mit einer Sprache für die Modellierung reaktiver Systeme mit den Konzepten von Zustandsmaschinen gezeigt. Die Implementierung des Ansatzes erfolgt durch das Eclipse-basierte Framework DMX und für die Simulationsbasissprache DBL. Dazu wurde ein adaptiver Modelleditor für die Basissprache entwickelt, der eine Syntaxerweiterung für Erweiterungsdefinitionen zur Programmlaufzeit des Editors durchführt. Damit lassen sich Erweiterungen auf eine ähnliche Art und Weise wie Funktionen definieren und verwenden. Die Konzeptreduktion wird durch einen ERL-Java-Transcompiler realisiert, der das Eclipse-basierte Metamodellierungs-Framework EMF verwendet. Die Ausführung von Modellen in der Basissprache DBL erfolgt durch einen DBL-Java-Transcompiler. Das Framework DMX und ein Simulationskern, der als Grundlage für einen DBL-C++-Transcompiler dienen kann, sind auf Github (http://github.com/ablunk/dmx) öffentlich zugänglich. Dabei wurde ein möglicher DBL-C++-Transcompiler prototypisch im Rahmen der Diplomarbeit von Chrisopher Breszka [17] implementiert.

6.1 Beiträge

Während der Arbeit an der Dissertation habe ich fünf wissenschaftliche Beiträge auf internationalen Konferenzen veröffentlicht und jeweils in einem Vortrag vor Ort verteidigt. Die Arbeiten wurden im Rahmen von Peer-Reviews auf ihre Qualität und Relevanz geprüft. Dabei handelt es sich um die folgenden Beiträge, die ich mit ihrem Titel, der Konferenz, dem Ort der Konferenz, dem Erscheinungsjahr und ihrer Zuordnung zu einem Kapitel der Arbeit angebe.

1. Prototyping Domain Specific Languages as Extensions of a General Purpose Language [11], SAM 2012: System Analysis and Modeling: Theory and Practice, Innsbruck, Österreich, 2012, Kapitel 5

2. Comparison of SLX and Model-Driven Language Development for Creating Domain-Specific Simulation Languages (Poster) [10], Winter Simulation Conference, Berlin, Deutschland, 2012, Kapitel 3

3. Efficient Development of Domain-Specific Simulation Modelling Languages and Tools [12], SDL 2013: Model-Driven Dependability Engineering, Montreal, Kanada, 2013, Kapitel 5

4. A Highly Efficient Simulation Core in C++ [13], DEVS '14 – Symposium on Theory of Modeling & Simulation, Tampa, Vereinigte Staaten von Amerika, 2014, Kapitel 4

5. Prototyping SDL Extensions [14], SAM 2014: System Analysis and Modeling: Models and Reusability, Valencia, Spanien, 2014, Kapitel 5

6.2 Ausblick

6.2.1 Namensauflösung

Sprachen verwenden zur Identifizierung von Sprachelementen Bezeichner, die an verschiedenen Stellen in einem Programm benutzt werden können. Auf diese Weise kann ein Sprachelement eine Beziehung zu einem anderen Sprachelement herstellen. Für die Verknüpfung der Sprachelemente über Bezeichner müssen Regeln festgelegt werden. Als Teil der Sprachdefinition von DBL sind diese Regeln im Framework DMX mit Hilfe Java festgelegt. Dazu wird durch den DBL-Modelleditor ein Java-Interface implementiert, das eine Operation für die Identifizierung eines Sprachelementes über einen Namen im Kontext eines anderen Sprachelementes definiert.

Erweiterungen können ebenfalls Bezeichner für Sprachelemente definieren. Im Rahmen dieser Arbeiten wurde nicht untersucht, wie die Identifizierung von Bezeichnern in Erweiterungen durch Regeln definiert werden kann. Eine Möglichkeit ist die Einführung einer weiteren Metasprache, wie z.B. der *Name Binding Language* [40], um Regeln für diese Identifizierung festzulegen.

6.2.2 Debugger

Ein wichtiger Bestandteil jeder Sprache ist ein Debugger, der die Unterbrechung der Ausführung eines Programms und die Betrachtung des Laufzeitzustandes erlaubt. Die Ableitung eines Debuggers aus einer Debugging-Beschreibung ist für metamodellbasierte Sprachen bereits untersucht [15]. Im Rahmen der vorliegenden Arbeiten ist eine Untersuchung hinsichtlich einer Übertragbarkeit auf eine Sprache, die durch Spracherweiterung definiert ist, interessant. Dabei müssen zwei Hauptprobleme gelöst werden: 1) die Beschreibung eines Laufzeitzustandes und 2) die schrittweise Fortführung der Ausführung.

Der Laufzeitzustand eines Programms ist abhängig von der Art der Sprache in der das Programm geschrieben ist. Bei einer objektorientierten Sprache setzt sich der Laufzeitzustand aus Threads, Stack-Frames, Variablen und einer Ausführungsposition zusammen. Bei einer Zustandsmaschinensprache besteht der Laufzeitzustand dagegen aus den laufenden Zustandsmaschinen, der Belegung der Signalempfangspuffer, dem aktuellen Zustand und der aktuell ausgeführten Transition oder Anweisung als Teil eines Zustandsübergangs.

Der Laufzeitzustand könnte durch eine Abbildung auf eine geeignete Objektstruktur im Basisprogramm dargestellt werden. Die Semantikdefinition könnte eine spezielle Debugger-Abbildung beschreiben, die Anweisungen für die Aktualisierung der Objektstruktur für den Laufzeitzustand erzeugt.

Für die Realisierung einer Steuerung der Zielprogrammausführung sehe ich zwei Möglichkeiten, die zu untersuchen sind:

1. *Anreicherung des Zielprogramms*: Das Zielprogramm wird mit Anweisungen angereichert, die die Ausführung nach jedem Schritt anhalten und durch Kommunikation mit dem DBL-Debugger die weiteren Aktionen abstimmen. Dazu muss eine Schnittstelle definiert werden, über die sich auch der Programmzustand erfragen lässt. Da Zielprogramm und Debugger in unterschiedlichen Betriebssystemprozessen laufen, muss ein Mechanismus zur Interprozesskommunikation verwendet werden.

2. *Verwendung eines Zielsprachen-Debuggers*: Der Zielsprachen-Debugger wird über ein bereitgestelltes Interface angesteuert. Über den Debugger lässt sich das Zielprogramm indirekt steuern und der Programmzustand lässt sich erfragen.

6.2.3 Trennung von Modellierungs- und Sprachverarbeitungskomponenten

Die bestehenden Umgebungen für die Entwicklung von Sprachen besitzen den Nachteil, dass die entwickelte Sprache immer an eine bestimmte Software für die Modellierung gebunden ist. Erweiterungen in SLX lassen sich nur in der SLX-Umgebung entwickeln, einem Programm, das nur unter Windows ausgeführt und nicht angepasst oder erweitert werden kann.

In MPS lassen sich Sprachen ebenfalls nur in der von JetBrains bereitgestellten MPS-Umgebung entwickeln. Die Metasprachen sind in MPS fest in die Entwicklungsumgebung integriert. Auf der einen Seite ist die Entwicklung von Sprachen dadurch einfacher, da eine feste Menge von Metasprachen bereitsteht. Auf der anderen Seite lassen

sich andere Metasprachen für bestimmte Teilaspekte nur schwer einsetzen. MPS bietet jedoch den Vorteil, dass der Quellcode der Software frei zugänglich ist und die MPS-Entwicklungsumgebung durch Plugins angepasst und erweitert werden kann.

Xtext verwendet Eclipse als Entwicklungsumgebung und ist auf Ecore als Metasprache festgelegt. Andere Sprachaspekte lassen sich jedoch in frei wählbaren Metasprachen beschreiben. Die Eclipse-Umgebung ist durch Plugins ebenfalls gut erweiterbar. Neben einer Eclipse-Integration befindet sich auch eine serverseitige Verarbeitung von Sprachinstanzen in Entwicklung, die auf dem *Language Server Protocol*[1] (LSP) aufbaut. Mit LSP ist es möglich die Modellierungskomponenten zu denen der Editor gehört von den Komponenten zur Verarbeitung einer Sprachinstanz zu trennen. Der Editor kommuniziert dazu mit einem Server. Er sendet den Programmtext bei Änderungen an den Server und erhält als Antwort ein Programm, das eine Hervorhebung der Syntax und Markierungen für Syntaxfehler enthält. Durch diese Trennung lässt sich die Editorkomponente leicht austauschen. Darüber hinaus ist es möglich Editoren für verschiedene Umgebungen wie z.B. das Web, die Cloud oder den Desktop bereitzustellen.

Die Verwendung des *Language Server Protocol* ist auch für DMX sinnvoll. Im Gegensatz zu Xtext besteht bei DMX die Schwierigkeit, dass die Editorkomponente erweiterbar sein muss. Hier wäre zu untersuchen, ob dies auf Basis von LSP und einem Editor, der LSP implementiert, realisiert werden kann.

[1] https://langserver.org

A Anhang

A.1 Anbindung der Basissprache an C++

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <boost/intrusive_ptr.hpp>
#include <boost/smart_ptr/intrusive_ref_counter.hpp>
#include <cstdint>

class String;

class Object : public boost::intrusive_ref_counter<Object, boost::thread_unsafe_counter> {
public:
    Object() {}
    virtual ~Object() {}

    virtual boost::intrusive_ptr<String> toString() const;

    virtual bool equals(boost::intrusive_ptr<Object> other) const {
        return this == other.get();
    }

    virtual int hashCode() const {
        return reinterpret_cast<uintptr_t>(this);
    }
};


class String : public Object {
private:
    std::string value;

public:
    String(std::string value) : value(value) {}

    std::string getValue() const {
        return value;
    }
};

boost::intrusive_ptr<String> Object::toString() const {
    std::string address = std::to_string(reinterpret_cast<uintptr_t>(this));
    std::string typeName = typeid(this).name();
    return new String(typeName + "/" + address);
}

template<typename T>
```

```
47 std::ostream& operator<< (std::ostream& stream, const boost::intrusive_ptr<T>& object) {
48     stream << object->toString()->getValue().c_str();
49     return stream;
50 }
51
52 class Iterator : public Object {
53 public:
54     virtual bool hasNext() = 0;
55     virtual boost::intrusive_ptr<Object> getNext() = 0 ;
56     virtual void remove() = 0 ;
57 };
58
59 class List : public Object {
60 public:
61     virtual void add(boost::intrusive_ptr<Object> object) = 0;
62     virtual boost::intrusive_ptr<Object> get(int i) = 0;
63     virtual int size() = 0;
64     virtual boost::intrusive_ptr<Iterator> iterator() = 0;
65 };
66
67 class ArrayListIterator : public Iterator {
68 private:
69     std::vector<boost::intrusive_ptr<Object> >::iterator iterator;
70     std::vector<boost::intrusive_ptr<Object> >& vector;
71 public:
72     ArrayListIterator(std::vector<boost::intrusive_ptr<Object> >& vector)
73         : iterator(vector.begin()), vector(vector) {}
74
75     virtual bool hasNext() {
76         return iterator != vector.end();
77     }
78
79     virtual boost::intrusive_ptr<Object> getNext() {
80         boost::intrusive_ptr<Object> current = *iterator;
81         iterator++;
82         return current;
83     }
84
85     virtual void remove() {
86         iterator = vector.erase(iterator);
87     }
88 };
89
90 class ArrayList : public List {
91 private:
92     std::vector<boost::intrusive_ptr<Object> > vector;
93
94 public:
95     void add(boost::intrusive_ptr<Object> object) {
96         vector.push_back(object);
97     }
98
99     boost::intrusive_ptr<Object> get(int i) {
100         return vector[i];
101     }
102
103     int size() {
```

```
104         return vector.size();
105     }
106
107     virtual boost::intrusive_ptr<Iterator> iterator() {
108         return new ArrayListIterator(vector);
109     }
110 };
```

Listing A.1: C++-Zwischenklassen für das in DBL als extern definierte Interface ArrayList inklusive aller abhängigen extern definierten Interfaces.

A.2 Laufzeitsystem für Prozesskontextwechsel in C++

```
1  class Execution {
2  public:
3      static int nextPid;
4      int pid;
5      int priority;
6      Execution* nextMoving; // next execution if this execution is in the moving list
7      MovingList* ml; // the moving list of this execution
8
9      Execution(int pid, int priority) : pid(pid), priority(priority) {
10         log2("created execution: pid = " << pid << ", priority = " << priority);
11     }
12
13     Execution(int priority) : pid(nextPid++), priority(priority) {
14         log2("created execution: pid = " << pid << ", priority = " << priority);
15     }
16
17     virtual void addAsNext(MovingList* rml, Execution* e);
18
19     // gets the moving list back into a consistent state if the first Execution is a DummyExecution,
20     // i.e. the moving list is essentially empty. this saves condition checks on the moving list,
21     // i.e. something like "first == NULL" checks.
22     virtual void updateMovingList();
23
24     virtual void eraseEmptyMovingList(Scheduler* sched);
25
26     // free resources needed for execution after the execution has terminated.
27     virtual void freeResources();
28 };
29
30 class DummyExecution : public Execution {
31 public:
32     DummyExecution(int priority) : Execution(-1,priority) {
33         nextMoving = this;
34     }
35     virtual void addAsNext(MovingList* rml, Execution* e);
36     virtual void updateMovingList();
37     virtual void eraseEmptyMovingList(Scheduler* sched);
38 };
39
40 void Execution::addAsNext(MovingList* rml, Execution* e) {
41     ml = rml;
```

```
42      nextMoving = e;
43      rml->last = e;
44      e->nextMoving = &(rml->dx);
45  }
46
47  void Execution::updateMovingList() {
48      // nothing to do for Execution objects
49  }
50
51  void Execution::eraseEmptyMovingList(Scheduler* sched) {
52  }
53
54  void Execution::freeResources() {
55      // nothing to do
56  }
57
58  void DummyExecution::addAsNext(MovingList* rml, Execution* e) {
59      e->ml = rml;
60      rml->first = e;
61      rml->last = e;
62      e->nextMoving = &(rml->dx);
63  }
64
65  void DummyExecution::updateMovingList() {
66      ml->last = this;
67  }
68
69  void DummyExecution::eraseEmptyMovingList(Scheduler* sched) {
70      sched->pml.erase(priority);
71      log2("erased moving list for priority=" << priority);
72  }
```

Listing A.2: Die Klasse Execution.

```
1   class Frame {
2   public:
3       void* dummy;
4       void* returnPoint; // pointer to return label
5   };
6
7   class GotoExecution : public Execution {
8   public:
9       void* cont; // continuation instruction
10
11      char mem[STACK_SIZE];
12      char* top;
13
14      // an active function call places its return value in the variable lrv.
15      union lrv_union {
16          int iv;
17          double dv;
18          bool bv;
19          void* pv;
20      } lrv;
21
22      // memory layout
23      // ============
```

```
24      // 0: values (actions part)
25      // top0 -> i: Frame object (stack frame 1)
26      // j: values (stack frame 1)
27      // top1 -> k: (free space)
28      void push(void* returnPoint, int vsize) {
29          log2("current top @ " << ((void*) top));
30          ((class Frame*)(top))->returnPoint = returnPoint;
31          top = top + sizeof(class Frame) + vsize;
32          log2("new top @ " << ((void*)top));
33          log2("pushed stack frame");
34      }
35
36      void pop(int vsize) {
37          top = top - sizeof(class Frame) - vsize;
38          log2("popped stack frame");
39      }
40
41      GotoExecution(int priority, void* cont) : cont(cont), Execution(priority) {
42          top = mem;
43          log2("mem @ " << ((void*)mem) << " to " << ((void*)(mem + STACK_SIZE)) );
44          log2("top @ " << ((void*)top));
45      }
46
47      virtual void freeResources() {
48          Execution::freeResources();
49          cont = NULL;
50          log2("pid-" << pid << ": freed resources.");
51      }
52  };
```

Listing A.3: Die Klasse GotoExecution und die Klasse Frame.

```
1   class MovingList {
2   public:
3       DummyExecution dx; // dummy execution for avoiding NULL pointer checks
4       Execution* first; // the current priority class as a linked list
5       Execution* last;
6
7   public:
8       MovingList(int priority) : dx(priority), first(&dx), last(&dx) {
9           dx.ml = this;
10      }
11
12      void yield();
13      void terminate(); // terminate current
14  };
15
16  // moves the current Execution to the end and returns the new first Execution
17  void MovingList::yield() {
18      Execution* cx = first; // the new current execution is the first execution in the moving list
19      first = first->nextMoving;
20      first->updateMovingList();
21      last->addAsNext(this, cx); // adds the execution after the current last execution
22  }
23
24  void MovingList::terminate() {
25      Execution* cx = first; // the current execution is the first execution in the moving list
```

```
26      first = first->nextMoving;
27      first->updateMovingList();
28  }
```

Listing A.4: Die Klasse MovingList.

```
1   class Scheduler {
2   public:
3       // each priority-class is managed as a linked list in a priority-keyed map
4       std::map<int, MovingList*, std::greater<int> > pml;
5
6       void add(Execution* e);
7
8       Execution* next();
9       Execution* yield();
10      Execution* terminate(); // terminate current and return next from moving list
11  };
12
13  void Scheduler::add(Execution* e) {
14      MovingList* rml; // the moving list related to the given element
15
16      try {
17          rml = pml.at(e->priority);
18      }
19      catch (const std::out_of_range& r) {
20          rml = new MovingList(e->priority);
21          pml.insert( std::pair<int, MovingList*>(e->priority, rml) );
22      }
23      rml->last->addAsNext(rml, e);
24  }
25
26  Execution* Scheduler::next() {
27      MovingList* hml = pml.begin()->second;
28      return hml->first;
29  }
30
31  Execution* Scheduler::yield() {
32      // 1. step:
33      // the current execution has to yield in its own priority class
34      // (it may not have highest priority one anymore)
35      cx->ml->yield();
36
37      // 2. step:
38      // the next execution has to be picked from the highest priority class
39      // (which may be higher than the priority class of the active execution)
40      return next();
41  }
42
43  Execution* Scheduler::terminate() {
44      Execution* tx = cx;
45
46      cx->ml->terminate();
47
48      // when a moving list for a certain priority gets empty, it has to be erased. that is because
49      // the PML is ordered by priority and we need access to the highest priority list in next().
50      cx->ml->first->eraseEmptyMovingList(this);
51
```

```
52      tx->freeResources();
53
54      return next();
55  }
```

Listing A.5: Die Klasse Scheduler.

A.3 Erweiterungen

```
1   module stdlib;
2
3   interface JavaClass {
4     bindings {
5       "java" -> "java.lang.Class"
6     }
7   }
8
9   interface Object {
10    bindings {
11      "java" -> "java.lang.Object"
12    }
13    boolean equals(Object other);
14    int hashCode();
15    string toString();
16  }
17
18  interface String extends Object {
19    bindings {
20      "java" -> "java.lang.String"
21    }
22    new(string original) {}
23  }
24
25  interface SystemOut {
26    bindings {
27      "java" -> "hub.sam.dmx.semantics.javabridge.SystemOut"
28    }
29    static void print(string s);
30    static void println(string s);
31  }
32
33  interface Iterator {
34    bindings {
35      "java" -> "java.util.Iterator"
36    }
37
38    boolean hasNext();
39    Object next();
40    void remove();
41  }
42
43  interface List {
44    bindings {
45      "java" -> "java.util.List"
46    }
```

```
47
48    boolean add(Object e);
49    boolean add(int index, Object e);
50    void clear();
51    boolean contains(Object e);
52    int size();
53    Object get(int index);
54    int indexOf(Object e);
55    boolean isEmpty();
56    boolean remove(Object e);
57    Object set(int index, Object e);
58    Object array[] toArray();
59    Iterator iterator();
60  }
61
62  interface ArrayList extends List {
63    bindings {
64      "java" -> "java.util.ArrayList"
65    }
66  }
67
68  interface Iterable {
69    bindings {
70      "java" -> "java.util.Iterable"
71    }
72    Iterator iterator();
73  }
74
75  interface Queue extends Iterable {
76    bindings {
77      "java" -> "java.util.Queue"
78    }
79
80    void offer(Object element);
81    Object poll();
82    Object peek();
83    int size();
84  }
85
86  interface LinkedList extends Queue {
87    bindings {
88      "java" -> "java.util.LinkedList"
89    }
90  }
91
92  interface Map {
93    bindings {
94      "java" -> "java.util.Map"
95    }
96
97    Object get(Object key);
98    void put(Object key, Object value);
99    void remove(Object key);
100 }
101
102 interface HashMap extends Map {
103   bindings {
```

```
104     "java" -> "java.util.HashMap"
105   }
106 }
```

Listing A.6: Die Bibliothek stdlib mit Standard-Bindings für Java.

```
1 #import "dbl"
2 #import "stdlib"
3 #import "stdx/foreach-collections"
4 #import "stdx/forever"
5
6 module stdx;
```

Listing A.7: Die Bibliothek stdx mit den Standard-Erweiterungen For-Each und Forever.

```
1 #import "../dbl"
2 #import "../stdlib"
3
4 module stdx_forever;
5
6 extension Forever extends dbl SimpleStatement {
7   start Forever;
8   Forever -> "forever" body:LocalScopeStatement;
9 }
10
11 semantics for Forever {
12   expand "while (true) " body;
13 }
```

Listing A.8: Die Erweiterung Forever.

```
1 #import "../dbl"
2 #import "../stdlib"
3
4 module stdx_foreach_collections;
5
6 extension ForEach extends dbl LocalScopeStatement {
7   start ForEach;
8   ForEach -> "foreach" "(" statements : list Variable "in" collection : Expression ")"
9     body : LocalScopeStatement;
10 }
11
12 semantics for ForEach {
13   Variable itemVariable = statements.get(0) as Variable;
14
15   ID it;
16   ID item;
17   expand "Iterator " it " = " collection ".iterator();";
18   expand "while (" it ".hasNext()) {";
19   expand " Object " item " = " it ".next();";
20   expand   itemVariable " = " item " as " itemVariable.getClassifierType() ";";
21   expand  body;
22   expand "}";
23 }
```

Listing A.9: Die Erweiterung For-Each.

Literaturverzeichnis

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[2] Anne Angermann, Michael Beuschel, Ulrich Wohlfarth, and Martin Rau. *MATLAB - Simulink - Stateflow: Grundlagen, Toolboxen, Beispiele*. Oldenbourg Wissenschaftsverlag, 2007.

[3] B.J. Arnoldus. *An illumination of the template enigma: software code generation with templates*. PhD thesis, Technische Universiteit Eindhoven, 2010.

[4] Jonathan Bachrach, Keith Playford, and Chandler Street. The Java Syntactic Extender (JSE). In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 31–42, 2001.

[5] J. W. Backus, H. Stern, I. Ziller, R. A. Hughes, R. Nutt, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, and P. B. Sheridan. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability on - IRE-AIEE-ACM '57 (Western)*, pages 188–198, New York, New York, USA, 1957. ACM Press. URL: `http://portal.acm.org/citation.cfm?doid=1455567.1455599`, `doi:10.1145/1455567.1455599`.

[6] Steffen Bangsow. *Fertigungssimulationen mit Plant Simulation und SimTalk: Anwendung und Programmierung mit Beispielen und Lösungen*. Carl Hanser Verlag GmbH & Co. KG, 2008.

[7] Rimon Barr, Zygmunt Haas, and Robbert Van Renesse. Jist: Embedding simulation time into a virtual machine. *EuroSim Congress on Modelling and Simulation*, 2004. URL: `http://cobweb.cs.uga.edu/~maria/pads/papers/jist-030612-simtime.pdf`.

[8] Ryan Bigg, Oscar Del Ben, and Frederick Cheung. Ruby on Rails Guides. URL: `http://guides.rubyonrails.org/`.

[9] Graham Birtwistle. *Demos — a system for Discrete Event Modelling on Simula*. Springer-Verlag New York, 1979.

[10] Andreas Blunk and Joachim Fischer. Comparison of SLX and Model-Driven Language Development for Creating Domain-Specific Simulation Languages (Poster at Winter Simulation Conference 2012), 2012.

[11] Andreas Blunk and Joachim Fischer. Prototyping Domain Specific Languages as Extensions of a General Purpose Language. In Øystein Haugen, Rick

Reed, and Reinhard Gotzhein, editors, *System Analysis and Modeling: Theory and Practice - 7th International Workshop*, pages 72–87. Springer Berlin Heidelberg, 2012. URL: http://link.springer.com/chapter/10.1007/978-3-642-36757-1_5#.

[12] Andreas Blunk and Joachim Fischer. Efficient Development of Domain-Specific Simulation Modelling Languages and Tools. In Ferhat Khendek, Maria Toeroe, Abdelouahed Gherbi, and Rick Reed, editors, *SDL 2013: Model-Driven Dependability Engineering*, pages 163–181. Springer Berlin Heidelberg, 2013. URL: http://link.springer.com/chapter/10.1007%2F978-3-642-38911-5_10.

[13] Andreas Blunk and Joachim Fischer. A Highly Efficient Simulation Core in C++. In *DEVS '14 Proceedings of the Symposium on Theory of Modeling & Simulation*, DEVS '14, San Diego, CA, USA, 2014. Society for Computer Simulation International. URL: http://dl.acm.org/citation.cfm?id=2665008.2665013.

[14] Andreas Blunk and Joachim Fischer. Prototyping SDL Extensions. In Daniel Amyot, Pau Fonseca i Casas, and Gunter Mussbacher, editors, *System Analysis and Modeling: Models and Reusability*, pages 304–311, Valencia, Spain, 2014. Springer International Publishing. URL: http://link.springer.com/10.1007/978-3-319-11743-0_21, doi:10.1007/978-3-319-11743-0_21.

[15] Andreas Blunk, Joachim Fischer, and Daniel Sadilek. Modelling a Debugger for an Imperative Voice Control Language. In Rick Reed, Attila Bilgic, and Reinhard Gotzhein, editors, *SDL 2009: Design for Motes and Mobiles*, pages 149–164. Springer, 2009.

[16] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag Berlin Heidelberg, 2003.

[17] Christopher Breszka. *Ein Transcompiler für die Simulationssprache DBL*. Diplomarbeit, Humboldt-Universität zu Berlin, 2017.

[18] Kehsiung Chung, Janche Sang, and Vernon Rego. A performance comparison of event calendar algorithms: an empirical approach. *Software: Practice and Experience*, 23(10):1107–1138, 1993.

[19] Sergey Dmitriev. Language oriented programming: The next programming paradigm, 2004. URL: http://www.onboard.jetbrains.com/articles/04/10/lop/.

[20] Jake Donham and Nicolas Pouillard. Camlp4 and Template Haskell. In *ACM SIGPLAN Commercial Users of Functional Programming on - CUFP '10*, New York, New York, USA, 2010. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1900160.1900167, doi:10.1145/1900160.1900167.

[21] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads : Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 29–42, New York, 2006. ACM.

[22] Sebastian Erdweg, Stefan Fehrenbach, and Klaus Ostermann. Evolution of Software Systems with Extensible Languages and DSLs. In *IEEE Software, Special Issue on New Directions in Programming Languages*, 2014.

[23] Joachim Fischer, Eckhardt Holz, Andreas Prinz, and Markus Scheidgen. Tool-based language development. *Computer Networks*, 49(5):676–688, 2005.

[24] The Eclipse Foundation. Eclipse Modeling Framework (EMF). URL: `http://www.eclipse.org/modeling/emf/`.

[25] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[26] Martin Fowler. Using the Rake Build Language, 2014. URL: `https://martinfowler.com/articles/rake.html`.

[27] Ritzberger Fritz. RunCC - A Java Runtime Compiler Compiler, 2004. URL: `http://runcc.sourceforge.net/`.

[28] Ralf Gerstenberger. *ODEMx – Neue Lösungen für die Realisierung von C++-Bibliotheken zur Prozesssimulation*. Diplomarbeit, Humboldt-Universität zu Berlin, 2003.

[29] Debasish Ghosh. *DSLs in Action*. Manning, 2011. URL: `https://dl.acm.org/citation.cfm?id=1965333`.

[30] Geoffrey Gordon. A general purpose systems simulator. *IBM Systems Journal*, 1(1):18–32, 1962. `doi:10.1147/sj.11.0018`.

[31] Tony L. Hansen. *The C++ Answer Book*. Addison-Wesley, 1990.

[32] Keld Helsgaun. A Portable C++ Library for Coroutine Sequencing. Technical report, Roskilde University, 1990.

[33] Keld Helsgaun. jDisco – a Java framework for combined discrete and continuous simulation. In *Datalogiske Skrifter (Writings on Computer Science)*, 2001.

[34] James O. Henriksen. SLX: The X is for Extensibility. In *Proceedings of the 32nd Conference on Winter Simulation*, pages 183—-190. Society for Computer Simulation International, 2000.

[35] James O Henriksen. Taming The Complexity Dragon. In *Winter Simulation Conference 2006*, 2006. URL: `http://www.informs-sim.org/wsc06papers/henriksen.pdf`.

[36] Itemis. Xtext, 2016. URL: `http://www.eclipse.org/Xtext/`.

[37] Itemis. Xtend, 2017. URL: `http://www.eclipse.org/xtend/`.

[38] JetBrains. MPS - Meta Programming System. URL: `https://www.jetbrains.com/mps/`.

[39] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '10*, pages 444–463, New York, New York, USA, 2010. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1869459.1869497, doi:10.1145/1869459.1869497.

[40] Gabriël D.P. Konat, Vlad A. Vergu, Lennart C.L. Kats, Guido H. Wachsmuth, and Eelco Visser. The spoofax name binding language. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12*, page 79, New York, New York, USA, 2012. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2384716.2384748, doi:10.1145/2384716.2384748.

[41] Frank Kühnlenz and Joachim Fischer. A Language-centered Approach for Transparent Experimentation Workflows. In *Proceedings of the CSSim 2011-Conference on Computer Modelling and Simulation*, 2011.

[42] Michael R. Lackner. Toward a general simulation capability. In *Proceedings of the May 1-3, 1962, spring joint computer conference on - AIEE-IRE '62 (Spring)*, page 1, New York, New York, USA, 1962. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1460833.1460835, doi:10.1145/1460833.1460835.

[43] Eamonn Lavery, Malcolm Beaverstock, Allen Greenwood, and William Nordgren. *Applied Simulation: Modeling and Analysis Using FlexSim*. FlexSim Software Products, Inc., 2011.

[44] Tim Lechler and Bernd Page. DESMO-J: An Object Oriented Discrete Simulation Framework in Java. In *Proceedings of the 11th European Simulation Symposium*, Erlangen, 1999. SCS Publishing House.

[45] Tomita Masaru. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th international joint conference on Artificial intelligence*, pages 756–764, Los Angeles, 1985. URL: https://dl.acm.org/citation.cfm?id=1623625.

[46] Richard E. Nance. A History of Discrete Event Simulation Programming Languages. In *Proceedings of the Second ACM SIGPLAN*, pages 149—-175. ACM, 1993. URL: http://eprints.cs.vt.edu/archive/00000363/, arXiv:arXiv:1306.2982v1, doi:10.1145/154766.155368.

[47] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2016.

[48] OMG. Unified Modeling Language (UML) 2.4.1, 2011. URL: http://schema.omg.org/spec/UML/2.4/.

[49] OMG. Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), 2013. URL: http://www.omg.org/spec/ALF/1.0.1.

[50] OMG. Meta Object Facility (MOF) 2.4, 2014. URL: http://www.omg.org/spec/MOF/2.4.2/.

[51] OMG. Object Constraint Language (OCL) 2.4, 2014. URL: http://www.omg.org/spec/OCL/.

[52] OMG. Unified Modeling Language (UML) 2.5, 2015. URL: http://www.omg.org/spec/UML/2.5.

[53] Terence Parr. *The Definitive ANTLR 4 Reference*. O'Reilly UK Ltd., 2013.

[54] Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools - PPPJ '13*, page 165, New York, New York, USA, 2013. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2500828.2500846, doi:10.1145/2500828.2500846.

[55] PragmaDev. SDL-RT, 2013. URL: http://www.sdl-rt.org/.

[56] Andreas Prinz, Birger Møller-Pedersen, and Joachim Fischer. Object-Oriented Operational Semantics. In *International Conference on System Analysis and Modeling*. Springer, 2016.

[57] Daniel A Sadilek. *Test-Driven Language Modeling*. Sierke Verlag, 2011.

[58] Daniel A. Sadilek and Guido Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In *Model Driven Architecture – Foundations and Applications*, pages 63–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. URL: http://link.springer.com/10.1007/978-3-540-69100-6_5, doi:10.1007/978-3-540-69100-6_5.

[59] Miro Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Taylor & Francis Ltd., 2008.

[60] Markus Scheidgen. *Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000*. Diplomarbeit, Humboldt-Universität zu Berlin, 2004.

[61] Markus Scheidgen. Integrating Content Assist into Textual Modelling Editors. In *Modellierung 2008*, pages 121–131, Berlin, 2008.

[62] Markus Scheidgen. *Descriptions of Computer Languages based on Object-Oriented Meta-Modelling*. PhD thesis, Humboldt-Universität zu Berlin, 2009.

[63] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Proceedings of the 3rd European Conference on Model Driven Architecture-foundations and Applications*, ECMDA-FA'07, pages 157–171, Berlin, Heidelberg, 2007. Springer-Verlag.

[64] Martin Schmidt, Arif Wider, Markus Scheidgen, Joachim Fischer, and Sebastian von Klinski. Refactorings in Language Development with Asymmetric Bidirectional Model Transformations. In *SDL 2013: SDL 2013: Model-Driven Dependability Engineering*, pages 222–238. Springer, Berlin, Heidelberg, 2013.

[65] Elmar Schoch, Michael Feiri, Frank Kargl, and Michael Weber. Simulation of Ad Hoc Networks: ns-2 compared to JiST/SWANS. *Proceedings of the First International ICST Conference on Simulation Tools and Techniques for Communications Networks and Systems*, 2008. URL: `http://eudl.eu/doi/10.4108/ICST.SIMUTOOLS2008.3021`, `doi:10.4108/ICST.SIMUTOOLS2008.3021`.

[66] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag, 2008.

[67] Stephen A. Schuman and Philippe Jorrand. Definition mechanisms in extensible programming languages. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), pages 9–20, New York, NY, USA, 1970. ACM.

[68] Edgar H Sibley and Dougl A S W Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[69] Jürgen Sieben. *Oracle PL/SQL: Das umfassende Handbuch für Datenbankentwickler*. Rheinwerk Computing, 2017.

[70] Michael Soden. *Dynamische Modellanalyse von Metamodellen mit operationaler Semantik*. PhD thesis, Humboldt-Universität zu Berlin, 2015.

[71] Michael Soden, Hajo Eichler, and Markus Scheidgen. A Meta-Modelling Framework for Modelling Semantics in the Context of Existing Domain Platforms. Technical report, Department of Computer Science, Humboldt-Universität zu Berlin, 2006.

[72] Thomas A. Standish. Extensibility in programming language design. *ACM SIGPLAN Notices*, 10(7):18, jul 1975. URL: `http://portal.acm.org/citation.cfm?doid=987305.987310`, `doi:10.1145/987305.987310`.

[73] Bjarne Stroustrup. A set of c classes for co-routine style programming. Technical report, Murray Hill, New Jersey : Bell Laboratories, 1980.

[74] Bjarne Stroustrup and Jonathan Shopiro. A set of c++ classes for coroutine style programming. In *Proceedings of the USENIX C++ Workshop*, pages 417–439, Berkeley, California, USA, 1987. USENIX Assoc.

[75] Falko Theisselmann, Frank Kühnlenz, Carsten Krüger, Joachim Fischer, and Tobia Lakes. How to reuse and modify an existing land use change model? Exploring the benefits of language-centered tool support. In *24th International Conference on Informatics for Environmental Protection*. Shaker-Verlag, 2010.

[76] Jean G. Vaucher and Pierre Duval. A comparison of simulation event list algorithms. *Commun. ACM*, 18(4):223–230, April 1975. URL: http://doi.acm.org/10.1145/360715.360758, doi:10.1145/360715.360758.

[77] Markus Voelter. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.

[78] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.

[79] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12*, page 121, New York, New York, USA, 2012. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2384716.2384767, doi:10.1145/2384716.2384767.

[80] Guido Wachsmuth. A Formal Way from Text to Code Templates. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 109–123. Springer-Verlag, 2009. URL: http://link.springer.com/10.1007/978-3-642-00593-0_8, doi:10.1007/978-3-642-00593-0_8.

[81] M P Ward. Language Oriented Programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[82] Dorian Weber and Joachim Fischer. Static Syntax Validation for Code Generation with String Templates. In *SDL 2017: Model-Driven Engineering for Future Internet*, pages 18–29. Springer, 2017.

[83] Elias Weingärtner, Hendrik Vom Lehn, and Klaus Wehrle. A performance comparison of recent network simulators. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC'09, pages 1287–1291, Piscataway, NJ, USA, 2009. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=1817271.1817510.

[84] Jim Weirich. Rake. URL: https://github.com/ruby/rake.

[85] Wolverine Software. SLX Overview. URL: http://www.wolverinesoftware.com/SLXOverview.htm.

[86] Daniel Zingaro. Modern Extensible Languages. Technical Report SQRL Report 47, McMaster University, 2007.

Selbstständigkeitserklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 126/2014 am 18.11.2014, habe ich zur Kenntnis genommen.

Berlin, den 15. Dezember 2017

Andreas Blunk